T.C.
MALTEPE ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ YÜKSEK LİSANSI

PARALEL HESAPLAMA TEKNİĞİNİN MONTE CARLO SİMULASYONUNDA UYGULANMASI

Bahadır KARASULU

Yüksek Lisans Tezi

Tez Danışmanı
Yrd. Doç. Dr. Şahin UYAVER

İSTANBUL – 2006

T.C.
MALTEPE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ YÜKSEK LİSANSI

PARALEL HESAPLAMA TEKNİĞİNİN MONTE CARLO SİMULASYONUNDA UYGULANMASI

YÜKSEK LİSANS TEZİ

Bahadır KARASULU

Tez Danışmanı
Yrd.Doç.Dr. Şahin UYAVER

İSTANBUL – 2006

“Paralel hesaplama tekniğinin Monte Carlo simulasyonunda uygulanması”

Özet

Bu yüksek lisans tezi paralel hesaplama tekniğinin bilimsel çalışmalardaki geçerliliğini ve önemini gösterme amacındadır. Paralel hesaplama tekniğine ihtiyaç duyulma nedeni, günümüzde bilgisayarların oldukça gelişmiş olmasına rağmen özellikle birçok bilimsel çalışmaların tek bir işlemci tarafından gerçekleştirilebilmesinin çok uzun zaman alması ve bunların yüksek bilgisayar belleklerine ihtiyaç duyuyor olmasıdır. Bu tezin gayesi doğrultusunda, paralel hesaplama tekniği bilimsel önemi olan bir sistem için uygulanmıştır. Sentetik sistemlerden biyo-sistemlere kadar geniş bir alanı kapsayan polimer fiziğinden bir sistemin Monte Carlo simulasyonu paralel hesaplama ile gerçekleştirilmiştir.

Tez beş bölümden oluşmaktadır. Birinci bölümde konuya giriş ve problemin tasviri, ikinci bölümde Paralel Hesaplama tekniği anlatılmıştır. Üçüncü bölümde MPI (Mesaj Geçme Arayüzü) konusunu detayları ile anlatarak uygulamalarına yer verilmiştir. Dördüncü bölümde problemin detayları verilmiş yapılan paralel hesaplama anlatılmış ve elde edilen sonuçlar gösterilmiştir. Son bölümde ise elde edilen sonuçlar değerlendirilmiştir.

Bu tez 2006 yılında yapılmıştır ve 107 sayfadan oluşmaktadır.

ANAHTAR KELİMELER: MPI, Paralel Hesaplama, Monte Carlo Simulasyonu, LAM-MPI, MCMC, Grid (Izgara), ideal polimer zinciri.

“Application of parallel computing technique to Monte Carlo simulation”

Abstract

This thesis aims to show the importance and the validity of parallel computing in scientific researches. The reason for the need to use parallel computing technique is that, although today’s computers are very well developed, many scientific works are not possible to be accomplished in only one computing unit. In the framework of this thesis, the technique of parallel computing is applied to a scientific problem, which has an importance in the pure science. The Monte Carlo Simulation of a polimeric system is carried out by using parallel computing technique.

The thesis consists of five chapters. In the first chapter, an introduction and the description of the work are given. In the second chapter, parallel computing is explained. In the next chapter, message passing interface (MPI) is explained in some detail and some applications are given. The fourth chapter is for the problem, the work done and also for the results obtained. The last chapter discusses the results and makes the conclusion.

This thesis was written in 2006 and it consists of 107 pages.

KEYWORDS: MPI, Parallel Computing, Monte Carlo Simulation, LAM-MPI, MCMC, Grid, ideal polymer chain.

Teşekkürler

Bu çalışmada kullanılan hesaplama kaynakları(nın bir kısmı), TÜBİTAK ULAKBİM Yüksek Başarımlı Bilgisayar Merkezi ve 104M359 proje nolu TÜBİTAK Mühendislik Araştırma Projesi tarafından sağlanmıştır.

Tez danışmanım Yrd. Doç. Dr. Şahin UYAVER 'e yol göstericiliğinden dolayı minnettarım. Bu çalışma boyunca bana karşı hep iyi davrandı ve yardımcı oldu. Onunla çalışmak büyük bir deneyimdi. Ayrıca değerli Prof. Dr. İdris GÜMÜŞ ve Prof. Dr. Fuat İNCE hocalarıma da teşekkür etmek isterim. Bana hep destek oldular.

Bu çalışma boyunca ailem büyük bir sabırla beni her zaman destekledi. Babama, anneme ve kardeşime teşekkür ederim.

Acknowledgement

The computational resources used in this work are provided by the TÜBITAK ULAK-BIM High Performance Computing Center and also by TÜBİTAK Engineering Research Project No_104M359.

I am so grateful to my thesis advisor Yrd. Doç. Dr. Şahin UYAVER for guidance. He was very kind and nice to me and helped me through this work. Working with him was a great experience. I'd like to thank Prof. Dr. İdris GÜMÜŞ and Prof. Dr. Fuat İNCE. They supported me always.

My family has always supported me with great patience during this work. Thanks to my dad, mom and brother.

İçindekiler

Şekil Listesi

Tablo Listesi

Kısaltmalar Listesi

ANSI	**:**	Amerikan Ulusal Standartlar Enstitüsü (American National Standartization Institution)
API	**:**	Uygulama Programlama Arayüzü (Application Programming Interface)
BLAS	**:**	Temel Lineer Cebir Sistemi (Basic Linear Algebra System)
CPU	**:**	Merkezi İşlem Birimi (Central Processing Unit)
DMA	**:**	Doğrudan Bellek Erişimi (Direct Memory Access)
EP	**:**	Şaşırtıcı derecede paralel (Embrassingly Parallel)
FDDI	**:**	Fiber Dağıtılmış Veri Arayüzü (Fiber Distributed Data Interface)
FLOPS	**:**	Saniye başına kayan nokta işlemleri (Floating point operations per second)
GNU	**:**	Genel Halka Açık Lisans (Linux İşletim Sistemi kavramı)
HPF	**:**	Yüksek Başarımlı Fortran (High Performance Fortran)
I/O	**:**	Girdi/Çıktı sistemi (Input/Output system)
JDK	**:**	Java Geliştirme Seti (Java Development Kit)
LA Library	**:**	Lineer Cebir Kütüphanesi (Linear Algebra Library) [Örn. PLAPACK, SCALAPACK, PETSc gibi kütüphaneler]
LAM/MPI	**:**	Yerel Alan Çoklu-bilgisayarları (Local Area Multicomputers)
MC	**:**	Monte Carlo yöntemi (MC Method)
MCM	**:**	Monomer başına MC adımı (MC step per monomer)
MCMC	**:**	Markov Zinciri MC Yöntemi (Markov Chain Monte Carlo Method)
MIMD	**:**	Çoklu Komut, Çoklu Veri (Multiple Instruction, Multiple Data)
MISD	**:**	Çoklu Komut, Tek Veri (Multiple Instruction, Single Data)
MPE	**:**	MPI'in kullandığı grafik arayüzü (MP Envoirenment)
MPI	**:**	Mesaj-Geçme Arayüzü (Message Passing Interface)
MPICH	**:**	Mesaj-Geçme Arayüzü'nün bir uyarlaması [Chameleon] (Message Passing Interface : Chameleon)
MPMD	**:**	Çoklu Program Çoklu Veri (Multiple Program Multiple Data)
MPPs	**:**	Kitlesel Paralel İşlemciler (Massive Parallel Processors)

NP	**:**	İşlemci Sayısı (Number of Processors)
NUMA	**:**	Tekdüze-olmayan Bellek Erişimi (Non-uniform Memory Access)
NUMMON	**:**	Monomer sayısı (Number of Monomers)
OpenMP	**:**	Açık Mesaj-Geçme sistemi (Open Message Passing)
PVM	**:**	Paralel Sanal Makine (Parallel Virtual Machine)
ROMIO	**:**	MPI tarafından geliştirilen özel Girdi/Çıktı sistemi
RPI	**:**	İstek İlerleme Arayüzü (Request Progress Interface)
SAW	**:**	"Self-Avoiding Walk" yürüme yöntemi
SIMD	**:**	Tek Komut, Çoklu Veri (Single Instruction, Multiple Data)
SISD	**:**	Tek Komut, Tek Veri (Single Instruction, Single Data)
SMP	**:**	Simetrik Çoklu İşlemciler (Symetric Multiple Processors)
SPMD	**:**	Tek Program Çoklu Veri (Single Program Multiple Data)
SSI	**:**	Sistem Servisleri Arayüzü (System Services Interface)
UMA	**:**	Tekdüze Bellek Erişimi (Uniform Memory Access)
WAN	**:**	Geniş Alan Ağı (Wide Area Network)
WWW	**:**	Dünya çapında ağ (World Wide Web)

Terimler Listesi

Benchmark	**:**	Kıyaslama
Beowulf	**:**	Paralel hesaplama yapmak için oluşturulmuş özel bir PC kümesi (NASA BEOWULF Projesi bu kümelemenin ilk örneği olmuştur).
Boltzmann sabiti	**:**	$k_B = 1.38066 \times 10^{-23}$ J/K $k_B = 8.61738 \times 10^{-5}$ eV/K
Cache	**:**	Önbellek
Client	**:**	İstemci
Cluster	**:**	Küme (genelde yüksek hızlı bir ağ ile birbirine bağlı olarak hesaplama işine iştirak eden bilgisayar topluluğuna verilen isim)
Deadlock	**:**	Kilitlenme
Deamon	**:**	Artalan süreci
Grid	**:**	Izgara (genel anlamda yazılım ve donanım yoluyla kümelerin oluşturduğu sistem)
Idle	**:**	Hazırda bekleme
Latency	**:**	Gecikme (veya iletişimde başlangıç zaman hatası)
RUN	**:**	Bağımsız çalışma veya koşma
Running Average	**:**	Yinelenen Ortalama
Self-Avoiding Walk	**:**	Bir noktadan diğerine kendini asla kesmeden giden yolda yürüme
Server	**:**	Sunucu
Speedup	**:**	Hızlanma

Bölüm 1

Giriş

Bu Yüksek Lisans tezi paralel hesaplama tekniğinin bir uygulamasını içermektedir. Paralel hesaplama tekniğine ihtiyaç duyulma nedeni, günümüzde bilgisayarlar oldukça gelişmiş olmasına rağmen özellikle birçok bilimsel çalışmaların tek bir işlemci tarafından gerçekleştirilebilmesinin çok uzun zaman alması ve bunların yüksek bilgisayar bellekle-rine ihtiyaç duyuyor olmasıdır. Paralel hesaplamadaki düşünce, bir işin farklı kısımlarının eşzamanlı olarak yürütülerek işin daha kısa sürede bitirilmesini sağlamaktır. Bu sayede daha büyük sistemleri inceleme olanağı sağlanır. Diğer yandan çok uzun hesaplama süreleri gerektiren sistemlerin de incelenmesi mümkün olur.

Açık bir paralel hesaplamada, birden fazla işlemci kullanılır. Birden fazla işlemci kullanarak paralel hesaplamayı gerçekleştirebilmek için bazı kütüphaneler hizmete sunulmuştur ve bu kütüphaneler sürekli geliştirilmektedir. Bunlardan en yaygın olanlarından biri “MPI: Message Passing Interface” olup bir çok araştırma konusunda sıkça kullanılmaktadır.

Polimer fiziği içerdiği konuları ile biyolojik sistemlere yakınlık arzetmesi ile birlikte çoğunlukla karmaşık sistemleri inceleme gereksinimi duymaktadır. Bu sebeble polimerik sistemlerin incelemelerinde paralel hesaplama çoğunlukla gerekli olmaktadır. Bu incelemeler yapılırken Monte Carlo simulasyon tekniği sıkça kullanılmaktadır. Polimer fiziğinde teorik olarak en basit sistem “ideal polimer zinciri”dir ve bu sistem diğer daha karmaşık sistemlerin anlaşılmasında önem arz etmektedir.

1.1 Problemin tasviri

Paralel hesaplamanın uygulaması ideal polimer zincirinin Monte Carlo Simulasyonu’nda gerçekleştirilmiştir. İdeal polimer zinciri, N tane monomer biriminden oluşan ve birimler arasında hiçbir etkileşmenin olmadığı bir sistemdir. Monomerler arasındaki bağ uzunluğu

sabittir. Bu tür bir sistem teorik, deneysel ve simulasyon çalışmalarında çok kez incelenmiş ve sonuçlar yayımlanmıştır. Diğer daha karmaşık polimer sistemleri, monomerler arasında ve monomer ile ortam arasında etkileşmelerin dahil edildiği sistemlerdir. Bu sistemlerin bilgisayar destekli incelemeleri öz olarak ideal polimer zincirinin incelenmesi ile aynıdır.

Paralel hesaplamanın geçerliliğini ve önemini gösterebilmek için, ideal polimer zincirinin Monte Carlo simulasyonu seri olarak yapılır ve elde edilen sonuçlar teorik öngörü ile karşılaştırılır. Aynı problem için paralelleştirme yapılır ve paralel hesap ile elde edilen sonuçlar teorik öngörüler ile karşılaştırılır. Yapılan hesabın doğruluğu kontrol edildikten sonra paralel hesaptaki hızlanmaya bakılarak performans analizi yapılır.

Bölüm 2

Paralel hesaplama

Geleneksel olarak yazılımlar seri hesap yapmak üzere yazılırlar. Bu yazılımlar tek işlemci üzerinde çalıştırılır. Seri hesaba ait problemler seri komutlar ile işlemcinin biri diğerinden sonra gelecek şekilde bu komutları çalıştırmasıyla çözümlenirler.

Basitçe paralel hesaplama, hesaplamalı bir problemin çözümü için bir çok hesap kaynağının **eşzamanlı** (simultane) olarak kullanılmasıdır. Bu hesap kaynakları bir çok işlemciye sahip tek bilgisayar veya bir ağ üzerinden birbirine bağlı sınırlı sayıda bilgisayardan oluşan veya her iki sistemi de içeren bir şekilde olmaktadır[1].

Hesaplamalı bir problem için

- yapılacak işi eşzamanlı olarak çözebilmek için işin ayrık parçalara bölünebilmesi,
- bir çok program komutunun eşzamanlı olarak çalıştırılabilmesi,
- bir çok hesap kaynağı ile problemi çözmenin tek hesap kaynağı ile çözmekten daha az vakit alabilmesi

şeklindeki koşullar vardır.

Paralel hesaplamanın kullanımı için iki ana neden vardır: Bunlar zamandan kazanmak ve büyük problemleri çözmek olarak bilinir[2]. Bazı başka nedenler de vardır. Bunlar arasında

- yerel olmayan kaynakların avantajlarını elde etme, örneğin Internet veya geniş alan ağları (WAN) üzerinden hesap kaynaklarına ulaşmak,
- maliyeti azaltmak, örneğin daha “ucuz” hesap kaynaklarını bir süperbilgisayarın yerine kullanabilmek,

- bellek kısıtlarının üstesinden gelmek, örneğin tek bilgisayar sınırlı bellek kaynaklarına sahip olduğundan büyük problemler için bir çok bilgisayarın belleğini kullanmak

gibi nedenler sıralanabilir.

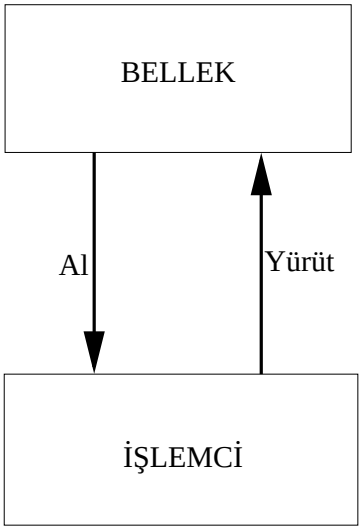
Paralel hesaplama gerçek hayattaki ilişkilerin durumlarına daima öykünen seri hesaplamanın bir evrimidir. Bu durumlar çok karmaşık ve birbirleriyle alakalı olarak aynı anda sıralı bir şekilde oluşur. Bazı örnekleri şunlardır: Gezegenlerin ve galaksinin gözlemlenmesi, hava ve okyanus örüntüleri, tektonik plaka hareketleri, metropollerde trafik, otomobil montaj hattı, bir iş yerindeki günlük işler.

Geleneksel olarak paralel hesaplama "hesaplamanın uç noktası" olarak bilinir. Paralel hesaplama karmaşık sistemlerin nümerik simulasyonu ve "Büyük Mücadele Problemleri" (Grand Challenge Problem) olarak bilinen, tek bilgisayarda çözümü imkansıza yakın problemlerle uğraşır. Bunlar hava ve iklim, kimyasal ve nükleer reaksiyonlar, biyolojik olaylar, insan genleri, jeolojik olaylar, sismik hareketler, mekanik aletler (uzay mekiğine kadar), elektronik devreler, üretim süreçleri olarak bilinir[3].

Ticari uygulamalar, bugün hızlı bilgisayarların üretilmesine ön ayak olmaktadırlar. Bu tarz uygulamalar karmaşık yollardan büyük miktarda verinin işlenmesini gerektirmektedir. Bunlara örnek olarak paralel veritabanları, veri madenciliği, petrol araştırmaları, web arama motorları, web tabanlı iş servisleri, tıpta bilgisayar destekli tanı, ulusal ve uluslararası firmaların yönetimi, eğlence sektöründe (kısmen) ileri grafik ve sanal gerçeklik uygulamaları, çoklu ortam teknolojileri ve ağ üzerinden video yayını verilebilir[4].

2.1 Seri ve paralel hesaplama

Kırk yıldan fazla süredir tüm bilgisayarlar Macar matematikçi John von Neumann'ın adıyla anılan genel bir makine modelinin takipçisi olmuşlardır. Şekil 2.1'de şematik olarak gösterilen bir von Neumann bilgisayarı saklanmış-program ana fikrini kullanır. İşlemci bellek üzerinde okuma ve yazma işlemi sıralarını belirten saklanmış bir programı çalıştırır[5]. Basitçe program ve veri komutlarını saklamada bellek kullanılır. Program komutları bilgisayara neler yapması gerektiğini belirten verileri kodlar. Veri programın kullandığı basit bilgilerdir. İşlemci komutları ve/veya veriyi bellekten alarak, komutları çözdükten sonra ardışımlı olarak onları icra eder.



Şekil 2.1: von Neumann seri bilgisayarı.

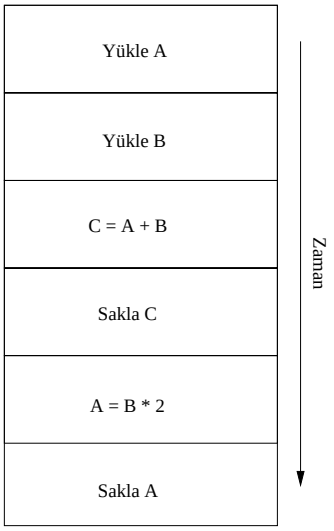
Paralel bilgisayarları sınıflandırmak için bir çok yol vardır. Bunlardan en yaygını 1966 yılından beri kullanılan Flynn taksonomisidir. Flynn taksonomisi

S I S D Tek Komut, Tek Veri	S I M D Tek Komut, Çoklu Veri
M I S D Çoklu Komut, Tek Veri	M I M D Çoklu Komut, Çoklu Veri

Şekil 2.2: Flynn taksonomisi.

Flynn taksonomisine ait 4 mümkün sınıflandırma Şekil 2.2'de verilmiştir. Bunlara sırayla bakacak olursak:

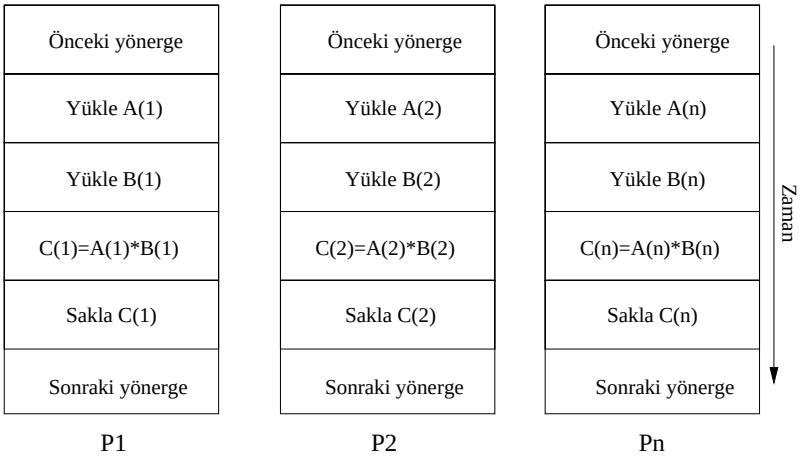
- **SISD (Tek Komut, Tek Veri)** sınıflandırmasında Şekil 2.3'de çalışma biçimi gösterilen seri bir bilgisayar ele alınır. Burada tek komut, anlam itibariyle herhangi bir saat çevrimi boyunca işlemci tarafından sadece bir komut akışı yapıldığında oluşur. Tek veri, anlam itibariyle sadece bir veri akışının herhangi bir saat çevrimi boyunca girdi olarak kullanılmasıdır. Bu sınıflandırmaya PC'ler veya tek işlemcili iş istasyonları örnek olarak verilebilir[5].



Şekil 2.3: SISD bilgisayarının çalışmasına örnek.

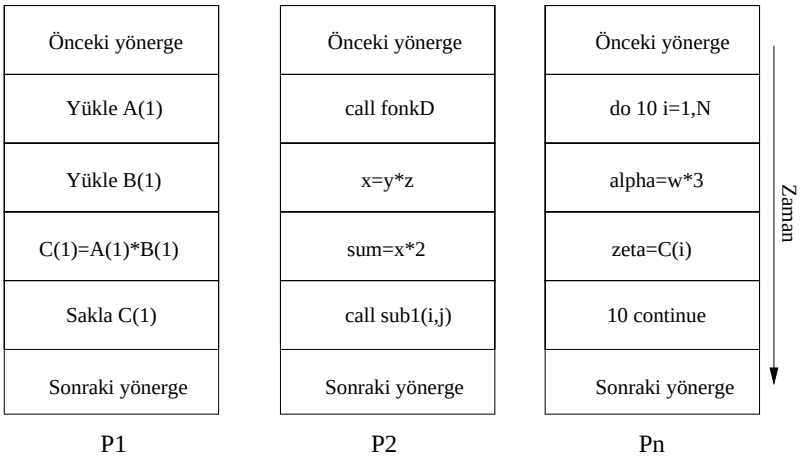
- **SIMD (Tek Komut, Çoklu Veri)** sınıflandırmasında Şekil 2.4'de çalışma biçimi gösterilen paralel bir bilgisayar ele alınır. Burada tek komut, anlam itibariyle tüm işlem birimlerinin verilen herhangi bir saat çevriminde aynı komutu çalıştırdıkları durumdur. Çoklu veri ise her işlem biriminin farklı bir veri elemanı üzerinde çalışması ile oluşur. Bu tip makineler tipik olarak bir komut göndericisine sahiptirler. Senkronize ve kesin bir yürütmeyle çalışırlar. Bu sınıflandırmadaki paralel makineler iki çeşittir: İşlemci dizileri ve vektör iş hatları. Örneğin işlemci dizileri için Connection Machine CM-2, Maspar MP-1 ve MP-2 söylenebilir. Ayrıca vektör iş-hatları için de IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi'nin makineleri örnek verilebilir[2].

- **MISD (Çoklu Komut, Tek Veri)** sınıflandırmasında birkaç paralel bilgisayar çeşiti vardır. Bunlar örneğin tek sinyal akışında çoklu frekans filtrelerinin çalıştırılması veya tek bir kodlanmış mesajın kırılmasının denendiği çoklu şifreleme (kriptoloji) algoritmalarının çalıştırılmasında kullanılırlar.



Şekil 2.4: SIMD bilgisayarının çalışmasına örnek.

- **MIMD (Çoklu Komut, Çoklu Veri)** sınıflandırması şu an en geçerli ve genel olarak kullanılan çeşittir. Şekil 2.5'de MIMD bilgisayarının çalışma biçimi görülmektedir. Bu tarz paralel bilgisayarlarda çoklu komut, anlam itibariyle her işlemcinin farklı bir komut akışını çalıştırabilir olduğu duruma denilmektedir. Çoklu veri ise her işlemci farklı bir veri akışı ile çalışabilir olduğu duruma denilir. Yürütme senkronize veya asenkronize olabilir. Örneğin bazı süperbilgisayarlar paralel bilgisayar ağları olan "ızgara" ve çok işlemcili SMP bilgisayarları bu tarza örnektir.

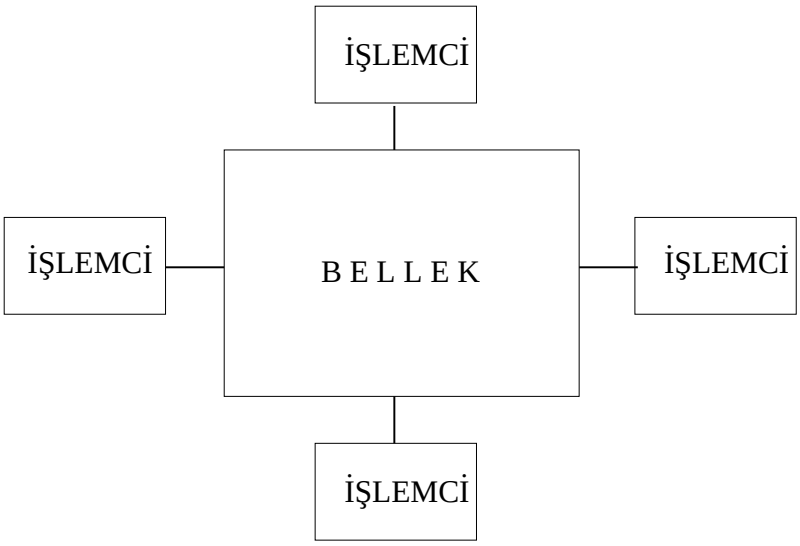


Şekil 2.5: MIMD bilgisayarının çalışmasına örnek.

2.2 Paralel bilgisayar bellek mimarileri

2.2.1 Paylaşımlı bellek

Genel karakteristik olarak paylaşımlı bellekli paralel bilgisayarlarda tüm işlemciler tüm belleğe global adres uzayı olarak erişebilirler. Bu mimari bütün işlemcilerin ortak bir belleği birbirleri ile paylaştığı bir durumdur. Burada işlemciler birbirinden bağımsız, fakat aynı bellek kaynağını paylaşacak şekilde çalışırlar. Bir işlemci tarafından değiştirilen bellek yerleşimi diğer işlemciler tarafından görülebilir durumdadır[4]. Paylaşımlı bellek mimarisi Şekil 2.6'da tasvir edilmektedir. Bu tarz makineler iki gruba ayrılır: UMA ve NUMA.



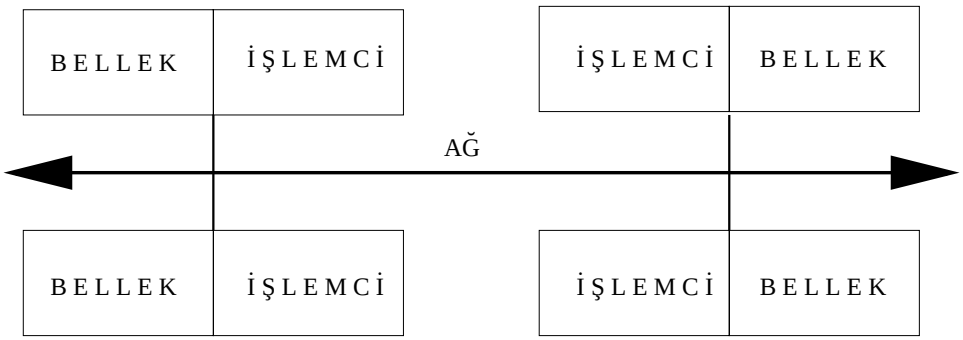
Şekil 2.6: Paylaşımlı bellekli mimari.

- **UMA (Tekdüze Bellek Erişimi):** En genel olarak günümüzün Simetrik Çoklu İşlemcileri (SMP) olan makineler buna örnektir. Bunlar eşit zamanlı ve eşit miktarda belleğe erişim yapan özdeş işlemcilerden oluşur.
- **NUMA (Tekdüze-olmayan Bellek Erişimi):** Genellikle fiziksel olarak birbirine bağlı iki veya daha fazla SMP makineden meydana gelir. Bir SMP makine başka bir SMP makinenin bellek adresine doğrudan erişebilir. Tüm işlemciler eşit erişim zamanları ile tüm belleklere erişmez. Erişim zamanları farklıdır. Bağlantılar üzerinden bellek erişimi yavaştır.

Global adres uzayı programlama için kullanıcı dostu bir bellek perspektifini sunmaktadır. Görevler arasında veri paylaşımı hem hızlı hem de tekdüzedir. Bu olay işlemciler için hafızanın yakın mesafede olmasından kaynaklanır. En önemli dezavantajı hafıza ve işlemciler arasındaki ölçeklenebilirlik sorunudur. İşlemci sayısı arttıkça paylaşımlı bellek ile işlemciler arasındaki trafik artar[3].

2.2.2 Dağıtık bellek

Paylaşımlı bellek sistemlerine benzer olarak dağıtık bellekli sistemler de bir çok çeşite sahiptir. Dağıtık bellekli sistemler işlemciler-arası belleğe bağlanabilmek (erişebilmek) için bir iletişim ağına ihtiyaç duyarlar. Şekil 2.7'de dağıtık bellekli mimari gösterilmektedir.



Şekil 2.7: Dağıtık bellekli mimari.

Her işlemci kendi yerel belleğine sahiptir. Bir işlemcideki bellek adresleri diğer bir işlemcideki bellek adreslerinden bağımsızdır. Bu sebeble işlemciler arasında global adres uzayı kavramı yoktur. Her işlemci kendi yerel belleğine sahiptir ve bağımsız olarak çalışır. Değişiklikler işlemcinin kendi yerel belleğinde olmakta, diğer işlemcilere etki etmemektedir. Eğer bir işlemci diğer bir işlemcideki veriye erişmek isterse, bu tamamen programcının sorumluluğunda ve programcının belirlediği veri iletişiminin "nasıl" ve "ne zaman" olması gerektiği konuları ile alakalı olarak ele alınmalıdır. Görevler arasındaki böyle bir iş ve senkronizasyon tamamen programcının sorumluluğundadır. Böyle sistemlerde veri transferi Ethernet gibi teknolojiler üzerinden yapılır. Bellek işlemci sayısı ile ölçeklenebilir. İşlemci sayısının artımı ile belleğin büyüklüğü orantılıdır. Engelleme olmadan her işlemci kendi yerel belleğine erişmektedir. Ayrıca programcı işlemciler arasındaki veri iletişiminin bir çok detayından da sorumludur[4].

2.2.3 Hibrid dağıtık-paylaşımlı bellek

Yukarıdaki sistemler haricinde Şekil 2.8'deki gibi hibrid dağıtık-paylaşımlı bellekli sistemlerde, paylaşımlı bellek bileşeni genellikle önbellekli SMP makinesidir. Verilen SMP makinesinde makinenin belleğini global olarak işlemciler adresleyebilmektedir. Aynı sistemde dağıtık bellek bileşeni SMP'lerin bir çoğunun oluşturduğu bir ağdır. SMP'ler kendi belleklerini bilmekte, fakat bir başkasınınkinden haberdar olmamaktadır. Böylece verinin bir SMP'den diğerine taşınması için iletişim ağına ihtiyaç duyulmaktadır.

Şekil 2.8: Hibrid dağıtık-paylaşımlı bellekli mimari.

2.3 Paralel programlama modelleri

Paralel programlama modellerinden bir kaçı şöyle sıralanabilir:

1. Paylaşımlı bellek modeli,
2. Kanallar ("Threads") modeli,
3. Mesaj geçme modeli,
4. Veri-paralel modeli,
5. Hibrid modeli.

Paralel programlama modelleri yukarıda açıklanan donanım ve bellek mimarilerinin bir özeti gibidir. Bu modeller belli bir makinaya veya bellek mimarisine özel değildirler. Prensip olarak (teorik olarak) modellerin hepsi her türlü donanıma uygulanabilir. Bu modellerin açıklamaları aşağıdadır.

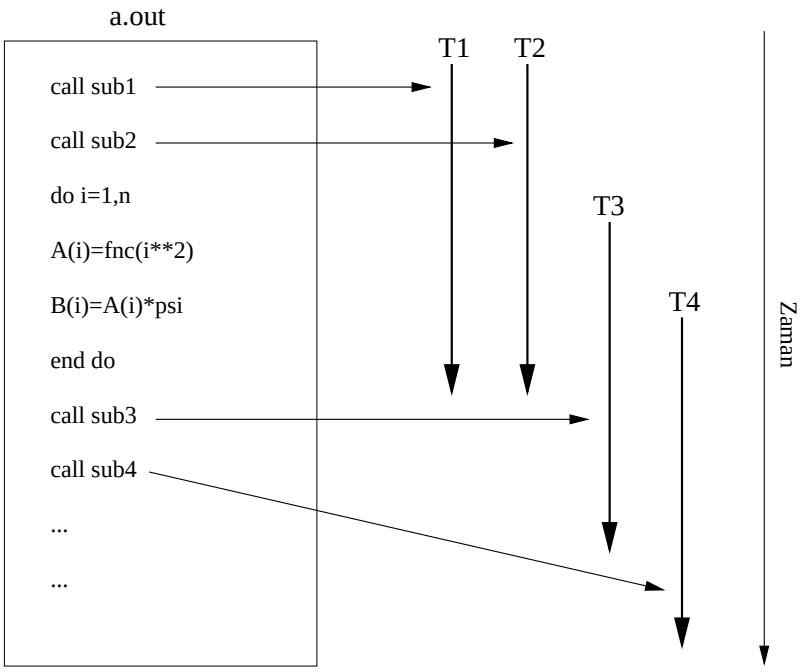
1. **Paylaşımlı bellek modeli:** Bu modelde görevler genel bir adres uzayını paylaşırlar, böylece okuma ve yazma asenkron olarak yapılır. Çeşitli mekanizmalar paylaşımlı belleğe erişimin kontrolünde kullanılır. Bu modelin programcıya sağladığı en büyük avantaj **"veri sahipliği"** konusudur. Görevler arasındaki veri iletişimini açıkça belirtmeye gerek olmaması programların üretilmesini kolaylaştırır.

2. **Kanallar modeli:** Bu paralel programlama modelinde, tek süreç çoklu ve uygun yürütme yollarına sahip olabilir. Altrutinleri de içeren tek bir program fikri bu modeli açıklamada yardımcı olmaktadır. Şekil 2.9'daki gibi bir **a.out** ana programını ele alırsak, işletim sistemi tarafından koşturulan program tüm gerekli sistemi ve kullanıcı kaynaklarını ele alır. Daha sonrasında **a.out** programı bu seri işleri icra ettikten sonra işlerin sayısını da (**kanal**) ayarlayarak işletim sistemi tarafından uygun olarak koşturulmasını sağlar. Her kanal yerel veriye sahiptir, fakat **a.out** programının tüm kaynaklarını paylaşırlar. Bu sayede her kanal için kopyalanan program kaynakları ile alakalı taşmadan korunulmuş olunur. Bir kanalın vazifesi en iyi olarak ana programda bir altrutin şeklinde tanımlanabilir. Herhangi bir kanal diğer kanallar gibi aynı zamanda herhangi bir altrutini çalıştırabilir. Kanallar birbirleriyle global bellek aracılığıyla iletişim kurarlar, bunu adres yerleşimlerini güncelleyerek yaparlar. Bu modelin uygulamaları olarak OpenMP ve POSIX kanalları sayılabilir.

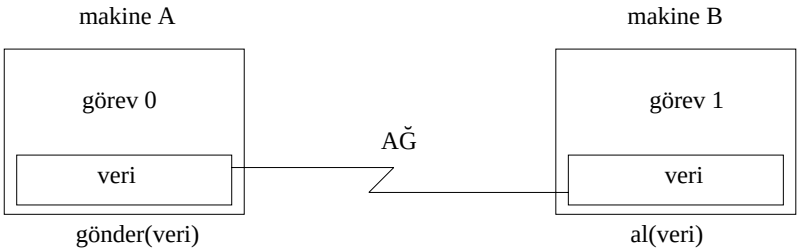
3. **Mesaj geçme modeli:** Bu model hesaplama boyunca kendi yerel belleklerini kullanan işler kümesine sahiptir[6]. Çoklu işler aynı fiziksel makine de olabileceği gibi, keyfi sayıda makinede de bulunabilir. İşlerin veri değiş-tokuşu sırasındaki iletişim, mesajlarının gönderilmesi ve alınması ile gerçekleştirilmektedir. Veri transferi genelde her bir süreç tarafından yapılması gereken toplu işlemlerle gerçekleştirilir. Şekil 2.10'da tasvir edilen bu model ayrıntıları ile tezin üçüncü bölümünde anlatılmaktadır.

4. **Veri-paralel model:** Bu modelde bir çok paralel iş bir veri kümesi üzerinde işlemler icra etmeye odaklanır. İşler kümesi aynı veri yapısı üzerinde ortaklaşa çalışır. Her iş (görev) aynı veri yapısının farklı bir bölümü üzerinde çalışır. Görevler aynı işlemi kendine ait bölüm üzerinde icra eder. Örneğin diziye bir eleman eklenilmesi durumu buna örnektir. Şekil 2.11'de gösterilen bu modelin uygulamalarına örnek olarak Fortran 90 ve HPF (Yüksek Başarımlı Fortran) ve derleyici direktifleri gösterilebilir. Derleyici direktifleri kaynak kod derlenirken verilen argümanlar olarak belirtilir[1].

5. **Hibrid model:** Modelin yapısı gereği iki veya daha fazla paralel programlama modeli harmanlanmıştır. Şu an kullanılan en güncel hibrid model örneklerinde mesaj

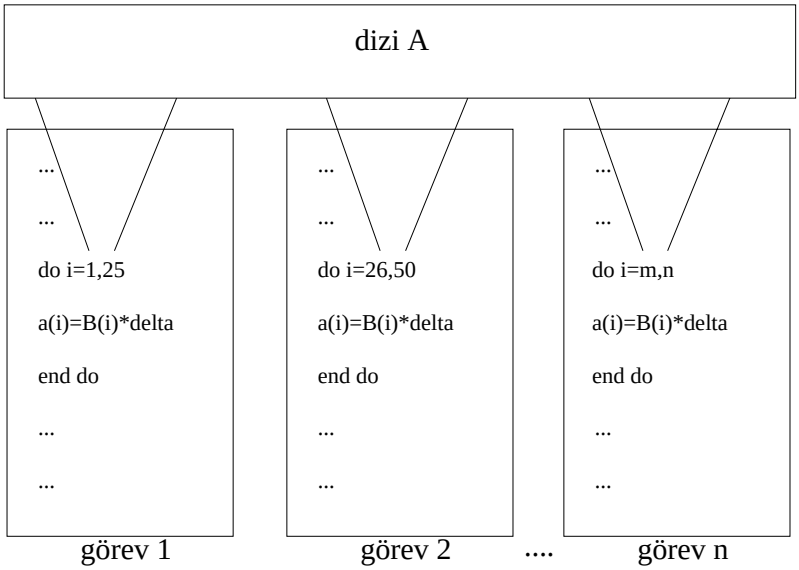


Şekil 2.9: Kanal modeli.



Şekil 2.10: Mesaj geçme modeli.

geçme modeli (MPI) ile kanal modeli (POSIX kanalları) veya paylaşımlı bellek modeli (OpenMP) harmanlanmaktadır. Böyle hibrid modeller ağ bağlantılı SMP makinelerine ait genel donanım çevresi üzerinde çalışmaktadır. Bir başka örnek ise veri-paralel model ile mesaj geçme modelinin harmanlanması ile oluşturulmuş olan modeldir. Bu model veri paralel uygulamaların (Fortran 90, HPF) dağıtık bellek mimarileri üzerinde çalıştırıldığı ve mesaj gönderip alma yöntemi uygulayan işlere sahip bir modeldir. Yukarıdakilere ilave olarak SPMD ve MPMD modelleri de vardır. **SPMD (Tek Program Çoklu Veri)** "yüksek seviyeli" programlama modeli yukarıda bahsedilen paralel programlama modellerinin harmanlanmasıyla oluştu-



Şekil 2.11: Veri-paralel modeli.

rulmuştur. Burada tek program tüm görevler tarafından eş zamanlı olarak yürütülür. Herhangi bir zamanda, işler aynı program içinden aynı veya farklı komutları çalıştırabilirler. SPMD programları genellikle gerekli mantıksal programlamaya farklı işleri ve dalları veya durumları programın sadece o parçasını çalıştırmak için sahiptirler. Görevlerin programın hepsini çalıştırmak yerine programın bir parçasını çalıştırması yeterli olmaktadır. **MPMD (Çoklu Program Çoklu Veri)** modeli de SPMD gibi "yüksek seviyeli" programlama modelidir. MPMD uygulamaları çoklu yürütülebilir obje dosyalarına (program) sahiptir. Uygulama paralel olarak çalışırken her iş diğer bir işe göre aynı veya farklı programı yürütebilir. Bu olay iş1 için **a.out** programı, iş2 için **b.out** programı vb. olarak düşünülebilir.

2.4 Paralel program tasarımı

Hiç şüphesiz paralel bir yazılımın üretilmesinin ilk adımı uygulaması yapılacak problemin tam ve net olarak anlaşılmasıdır. Böyle bir problemin paralel çözümünü üretmek için zaman harcamaya başlamadan evvel problemin gerçekten paralelleştirilebilir olup olmadığını belirlemek gerekir[7].

- **Paralelleştirilebilir probleme örnek:**
  "Bir molekülün bağımsız birkaç bin yapısının her biri için potansiyel enerjiyi he-

saplayın. Hesap bitince tüm yapının minimum enerjisini bulun."
Böyle bir problem, yapılar birbirinden bağımsız olarak ele alınabildiği için paralel yöntemle çözülebilir ve minimum enerji bulunabilir.

- **Paralelleştirilemeyen probleme örnek:**
  Fibonacci serisi (1, 1, 2, 3, 5, 8, 13, 21, ...) şeklindedir. Aşağıdaki formülle ifade edilir:

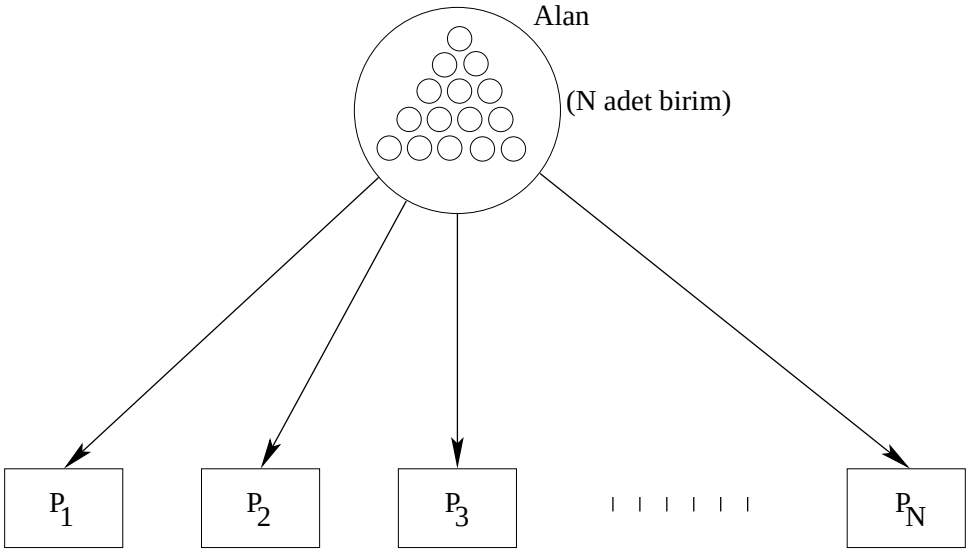
$$F(k+2) = F(k+1) + F(k) \tag{2.1}$$

  Bu serinin çözümünde paralel bir yaklaşım yapılamamaktadır. Çünkü serinin elemanları kendinden önce gelen eleman ile hesapsal olarak bağımlıdırlar. Paralelleştirme yapılabilmesi için birbirlerinden bağımsız olması gerekmektedir.

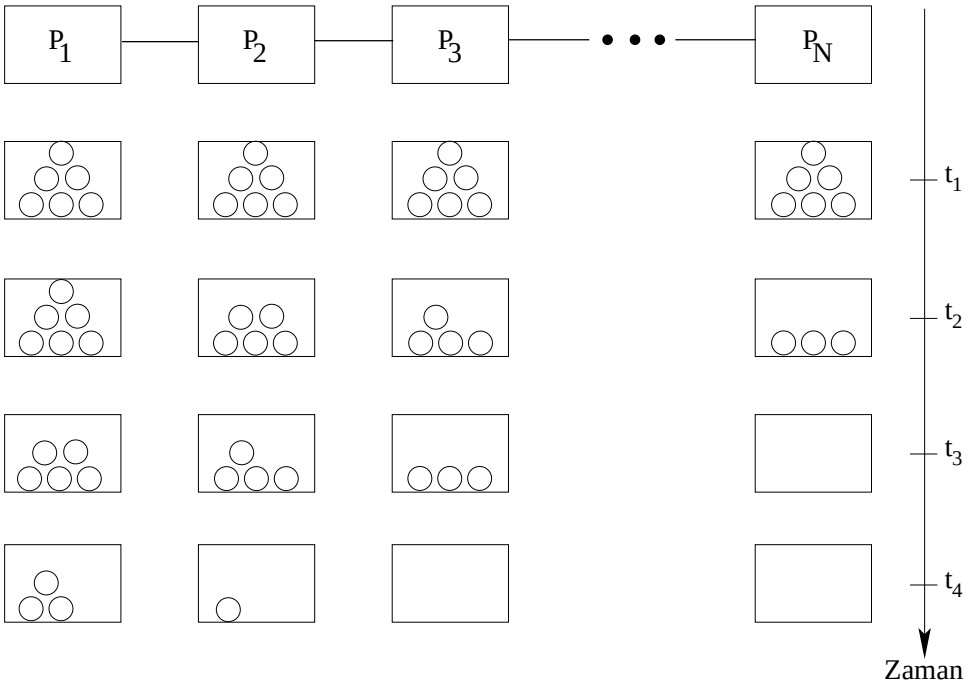
Programın **can alıcı** noktalarını tanımlamak gerekmektedir. Gerçekten istenilen işin doğru yapıldığından emin olmak gerekir. Bunu anlamak için profil çıkartmak ve performans analiz araçları kullanmak, işlemci kullanımı gibi unsurları göz önünde bulundurmak çok önemlidir. Buna rağmen bazı darboğazlar ile karşılaşılabilmektedir. Bu darboğazlar programın yavaş çalışması, işin tam yapılamaması, hatalı yapılması gibi sonuçlar doğurabilir. Bu noktada veri bağımlılığına (Fibonacci serisinde olduğu gibi) dikkat etmek gerekir[2].

Paralel bir programın tasarımındaki ilk adım problemi birbirinden *ayrık* **"parçalara" (chunks)** ayırma gerekliliğidir. Böylece bu parçalar çoklu görevlere dağıtılabilecektir. Bu durum decomposition (ayrışım) veya bölümleme olarak bilinir. Bu durumun örnekleri Şekil 2.12 ve Şekil 2.13'de görülmektedir. İki farklı temel ayrışım yolu bulunmaktadır: Alan ayrışımı ve Fonksiyonel ayrışım.

Alan ayrışımında problemle ile ilgili veri ayrıştırılır. Her paralel iş kendine ait parça üzerinde çalışır. **Fonksiyonel ayrışımda** hesap tarafından yönetilen veri ile icra edilecek olan hesaplamayla ilgilenilir. Problem yapılması zorunlu iş için ayrıştırılarak her bir görevin tüm işin bir parçasını icra etmesi sağlanır.



Şekil 2.12: Veri alanının her süreç için eşit boyutlu veri kümelerine ayrışımı.



Şekil 2.13: Süreçler için “yük” ’ün tekdüze dağılımı.

2.5 İletişim

Problemin cinsine göre görevler arası iletişim farklılık gösterebilir. Aşağıdaki gibi iki gruba ayrılabilir:

- **İletişime ihtiyaç bulunmayan durum:** Bazı problem tipleri bölümlenebilir (ayrıştırılabilir) ve veri paylaşımı için sanal olarak görevlere ihtiyaç duymadan paralel biçimde yürütülebilir. Örneğin bir resim işleme işleminde siyah-beyaz bir resimdeki her pikselin renginin tersine çevrileceği durumu ele alırsak, resim verisi çoklu işlere kolayca dağıtılarak tüm işin kendine ait parçasını yapan ve her biri birbirinden bağımsız hareket eden parçalara ayrılır. Bu tarz problemler genelde Embarrassingly Parallel (Şaşırtıcı derecede paralel) olarak adlandırılır. Görevler arası iletişime pek az ihtiyaç duyarlar.

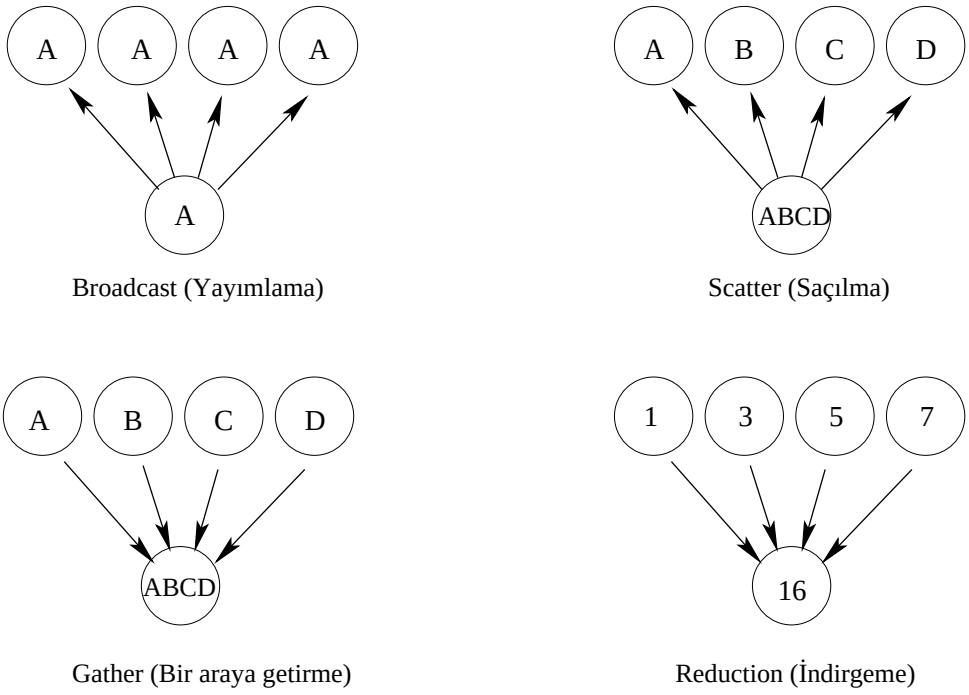
- **İletişime ihtiyaç bulunan durum:** Birçok paralel uygulama o kadar da basit değildir. Veriyi birbirleriyle paylaşmak için görevlere ihtiyaç duyarlar. Örneğin bir 3Boyutlu ısı yayılım problemi, komşu verilere sahip görevler tarafından hesaplanılmış ısıyı bilen bir göreve ihtiyaç duyar. Komşu verilerin değişimi bu görevin verisi üzerinde doğrudan etkili olur.

Program tasarımı sırasında görevler arası iletişime etki eden önemli faktörler vardır[4]. Bunlar şu şekilde sıralanabilir:

- **İletişim maliyeti:** Görevler arası iletişim daima sanal olarak taşmayı (yığılmayı) kapsamaktadır. Hesaplamada kullanılabilecek kaynaklar ve makine döngüleri iletilen veri ve paket yerine kullanılır. İletişimler sıklıkla görevler arasında bazı senkronizasyon (aynı duruma gelme) tiplerine ihtiyaç duyarlar. Bu senkronizasyon tipleri görevlerde iş yapmak yerine "bekleme" yaparak zaman harcanmasına neden olurlar. İletişim trafiğini arttırmak mümkün ağ bantgenişliğini doldurur; böylece başarım ile ilgili problemler daha da artar, trafik kötüleşir.

- **Gecikme ve Bantgenişliği: Gecikme (Latency)** asgari bir mesajı (sıfır byte) A noktasından B noktasına göndermek için geçen süredir. Genel olarak mikrosaniye cinsinden verilir. **Bantgenişliği (Bandwidth)** birim zaman başına iletişim yapılabilir veri miktarına denilir. Genel olarak megabyte/saniye cinsinden verilir. Bir çok küçük mesajın gönderilmesi iletişim yığılmasını arttırarak gecikmeye neden olur. Küçük mesajları büyük bir mesaj paketinin içine koymak genellikle daha verimlidir. Böylece etkin iletişim bantgenişliği artar.

- **İletişimin görünürlüğü:** Mesaj geçme modeliyle iletişim programcının kontrolü altında tamamen görünür ve kesindir. Veri-paralel modeliyle iletişim genellikle programcı için şeffaf olur (kısmen dağıtık bellek mimarilerinde de olur). Programcı görevler arası iletişimin nasıl kurulabildiğini tam olarak bilmeyebilir.



Şekil 2.14: İletişim çeşitleri.

- **Senkronize ve Asenkronize iletişimler:** Senkronize iletişimler bazı "el sıkışma" tiplerine veriyi paylaşan görevler arasında ihtiyaç duyarlar. Bu kesin olarak kodun içerisinde programcı tarafından yapılandırılmış olmalıdır veya programcı tarafından düşük seviyede bilinerek gerçekleştirilmelidir. **Senkronize iletişim** genelde **bloklanmış** iletişim olarak bilinir, çünkü iletişim tamamlanana kadar diğer görevler beklemek zorundadır. **Asenkronize iletişim** görevlerin birinden bir diğerine bağımsız olarak verinin aktarılmasına izin verir. Asenkron iletişim genellikle **bloklanmamış** iletişim olarak bilinir, çünkü iletişimler icra edilirken diğer görevler yapılabilmektedirler. İletişimde aralıklarla hesaplama asenkron iletişimin yegane avantajıdır.
- **İletişim sahası:** Hangi görevlerin birbirleriyle iletişimde bulunmak zorunda olduğunu bilmek, paralel bir kodun tasarım aşaması boyunca en kritik noktadır. Aşağıda

açıklanan iki saha, senkronize veya asenkronize duruma uygulanabilir: **Noktadan-noktaya iletişim** biri gönderici/verinin üreticisi ve diğeri alıcı/tüketici olarak davranan iki görevi içerir. **Toplu iletişim** topluca veya genel bir grubun üyeleri şeklinde tanımlanan iki veya daha fazla görev arasındaki veri paylaşımını içerir. Bu çeşit iletişim örnekleri Şekil 2.14'de görülmektedir.

- **İletişim verimi:** Hangi iletişim işleminin veya hangi modelin kullanıldığına bağlı olarak programcının kendi seçimleriyle başarım etkilenmektedir.

2.5.1 Senkronizasyon

Paralel hesaplamada bazen hesabın ve verinin diğer hesap ve verilerle karışmaması istenebilir. Bu amaçla işlemciler senkronize edilebilirler. Bazı senkronizasyon tipleri şöyle sıralanabilir:

- **Bariyer:** Genellikle tüm görevleri kapsar. Her görev bariyere varana kadar çalışır, varınca dururlar veya bloklanırlar. En son görev bariyere varınca, artık hepsi senkronize olmuş durumdadırlar[7].

- **Kilit/Semafor:** Herhangi bir sayıda görevi içerebilir. Genelde kodun bir bölümü veya global veriye erişimi serileştirmek (korumak) için kullanılır. Bir anda sadece bir görev kilit/semafor/bayrak olarak kullanılır (sahip olunur). İlk görev kilidi oluşturup ona erişirken, diğer görevler kilide erişmeye çalışsa da bu ilk görevi beklemek zorundadırlar. Ancak ilk görev kilidi serbest bırakınca diğerleri bu kilide sahip olabilir. İletişim bloklanmış veya bloklanmamış olabilir[8].

2.6 Yük dengeleme ve performans

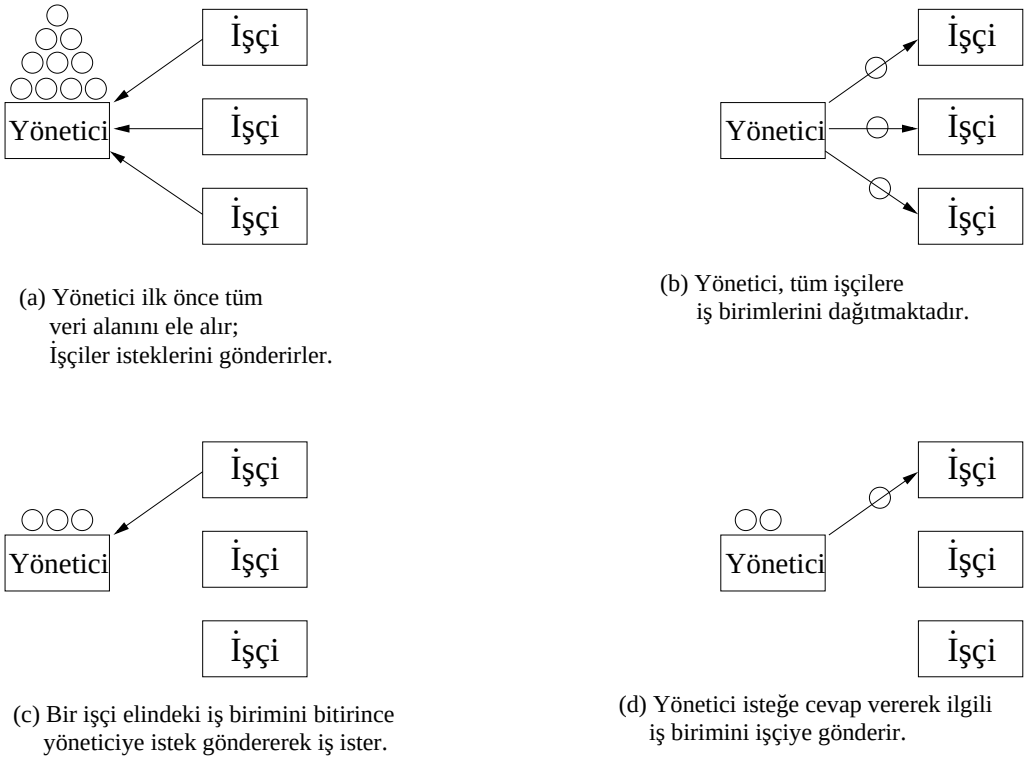
Yük dengeleme, işlemci kullanımını mümkün olduğunca yüksek düzeyde tutmayı sağlar. Statik yük dengeleme, hem programı çizelgeleme veya modelleme yoluyla hem de ağ'daki her işlemcinin kullanımını bilinçli olarak, yükleri ters orantılı olacak biçimde, işi yaklaşık aynı zamanda bitirmeyi garanti edebilecek işlemcilerin hepsine yükü atayarak gözlemlemeyi sağlar.

Bu yaklaşım iyi bir performans sağlar. Fakat ne yazık ki, çizelgeleme her zaman kolayca gerçekleştirilemez, ayrıca modelleme uygulamanın kontrol yapısının çok iyi bilinmesini gerektirir, buna veri alanı özellikleri ve donanım (iletişim performansı) bilgisinin eklenmesi gerekir[9].

Dinamik yük dengeleme yöntemi, statik yük dengelemedeki kadar etkin bir performans vermez. Fakat ilgili performansın avantajı süreçlerin yüklerini paylaşmalarıyla sağlanır. Dinamik yük dengelemenin bir çok çeşidi vardır, örneğin *Merkezi* ve *Dağıtılmış* gibi. *Yönetici-İşçiler* yöntemi tipik bir merkezi yük dengeleme şeması örneğidir ve sıklıkla çoklu işlemcilerin mesaj-geçme işlerinde kullanılır.

2.7 Yönetici-İşçiler yöntemi

Yönetici-işçiler uygulamasında bir süreç veri alanının bakımı ve daha fazla veri isteğinde bulunan diğer süreçlere bu verinin dağıtımı şeklindeki özel bir görevi üstlenir. Bu süreç işçileri sevk ve idare eden bir yönetici rolünü onları çalıştırmak için yapar. Şekil 2.15'de bu yöntem tasvir edilmektedir.



(a) Yönetici ilk önce tüm veri alanını ele alır; İşçiler isteklerini gönderirler.

(b) Yönetici, tüm işçilere iş birimlerini dağıtmaktadır.

(c) Bir işçi elindeki iş birimini bitirince yöneticiye istek göndererek iş ister.

(d) Yönetici isteğe cevap vererek ilgili iş birimini işçiye gönderir.

Şekil 2.15: Yönetici-işçiler yöntemi.

Yönetici-işçiler yaklaşımıyla dengelenmiş bir sisteme bakacak olursak, süreç makinelerinin oluşturduğu bir ağda bir süreç yönetici olarak veri alanını ele alır, çünkü veri alanı genellikle sunucu tarafından oluşturulur veya sonunda bu alan toparlanarak sunucuya geri

gönderilmek zorundadır. Yönetici burada **"kök"** düğüm vazifesi yapar. İşçiler yönetici-düğüme iş isteğini olabildiğince kısa sürede göndererek görevlerine başlarlar. Yönetici her işçiye iş birimlerini göndermekle yükümlüdür. İlerleyen zamanlarda bazı işçiler diğerlerinden önce işlerini bitirerek daha fazla iş isteyebilirler. Bu olayın yönetimi yönetici tarafından hesaplamanın ilerleyişi ve işlemci kullanımlarının seviyesine göre yük dağılımı ile ayarlanır[9].

2.8 Paralel hesaplamanın sınırları ve eniyileştirme

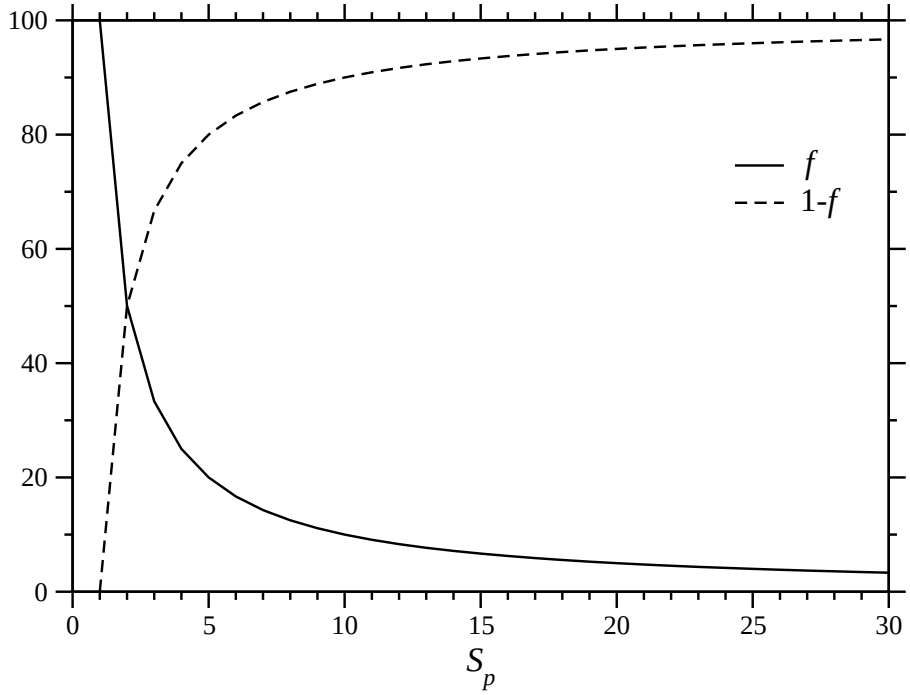
Hesaplamalı işi paralel olarak icra etmenin teorisinde hesabı paralel hale getirmekten elde edilebilecek kazancı sınırlayan temel kanunlar vardır. Bu kanunları anlamak için, öncelikle amacın ne olduğu tanımlanmalıdır. Genel olarak büyük ölçekli hesaplamalarda[10] amaç mevcut imkanlar çerçevesinde mümkün en kısa zaman içerisinde en fazla iş yapmaktır. Eğer biz aynı zaman içinde daha büyük iş yapabiliyor isek veya aynı işi daha kısa zamanda bitirebiliyor isek o zaman amacımıza ulaştığımız söylenebilir. Bu tarzda hesap yapan bir sistemin "gücü" yapılabilen işin bu işi yapmak için geçen zamana oranı şeklinde tanımlanabilir. Genellikle biz birim maliyet başına gücü eniyileştirmek isteriz (yani maliyet-kazanç ilişkisi) [1].

2.8.1 Amdahl kanunu ve paralel hızlanma

Paralel hesaplama ile alakalı en yaygın gözlem, herbir algoritmanın paralel bir bilgisayarda elde edilebilecek hızlanmayı sınırlayan sıralı bir kısmının olduğudur. Bu gözlem *Amdahl kanunu* olarak adlandırılır ve şu şekilde ifade edilir: Eğer bir algoritmanın sıralı bileşeni programın toplam yürütme zamanının f kadarı ise o zaman paralel bir hesaplamada elde edilebilecek maksimum hızlanma $1/f$'dir. Örneğin sıralı bileşen %5 ise, elde edilebilecek maksimum hızlanma 20'dir. Şekil 2.16'da gözüktüğü gibi, hızlanmanın yaklaşık 2 olduğu bir durumda hesabın yaklaşık %50'si paralelleştirilebilmiştir.

İlke olarak n_p işlemci (buradan sonra n_p=işlemci sayısı olarak geçecektir) arasında bölünen iş $1/n_p$ süresinde tamamlanacaktır. Bu hesaplama gücünün n_p-katlı olarak artması demektir. Fakat paralelleştirilmiş bir işin herhangi bir kısmı, tek bir işlemci tarafından seri olarak yapılması gereken kısımları içerecektir. Bu kısım paralel işlemciler üzerinde

[1] Fiziksel ve mali koşullar tek işlemcili sistemlerin gücünü sınırlar. Daha önce söylendiği üzere, tek işlemcili sistemlerin güçlerinin artışı genellikle çok zayıf ve lineer olmadığından dolayı ve hesap yapan bir sistemin gücünü arttırmanın bir yolu bir probleme birden fazla işlemciyi uygulamak olduğu gerçeği, *BEOWULF* tasarımının motivasyon kaynağıdır (BEOWULF için [11] kaynağına bakınız.)



Şekil 2.16: f ve $(1-f)$'nin hızlanma ile değişimi.

yürütülemez (yürütülebilse bile daha yavaş olacaktır). Sadece paralelleştirilebilen kısım n_p kere daha hızlı yürütülecektir.

Paralel bir hesabın hızlanması ("speed-up"), n_p işlemcide çalıştırıldığı z aman yapılan işin (yani güç) tek bir işlemcide yapılan işe oranı şeklinde tanımlanır. Konuyu basitleştirmek gayesi ile, yapılması gereken keyfi bir görev üzerine odaklanalım: O zaman biz hızlanmayı (n_p'nin bir fonksiyonu olarak güçteki artış) bu işi yapabilmek için tek bir işlemcide geçen sürenin n_p işlemcide geçen süreye oranı olarak tanımlayabiliriz.

$T(n_p)$, görevi n_p işlemcide tamamlamak için gerekli süre olsun. $S(n_p)$ hızlanması

$$S(n_p) = \frac{T(1)}{T(n_p)} \tag{2.2}$$

oranı olacaktır. Bir çok durumda $T(1)$ içinde T_s seri kısım ve T_p paralelleştirilebilen kısım bulunur. Paralel kısım bölünüp ayrıldığı zaman seri yürütme süresi yok olmaz. En ideal durumda, paralel yürütme süresi $1/n_p$ faktörü ile azalır. Bu tarzda hızlanma aşağıdaki şekilde tekrar yazılabilir:

$$S(n_p) = \frac{T(1)}{T(n_p)} = \frac{T_\mathrm{s} + T_\mathrm{p}}{T_\mathrm{s} + T_\mathrm{p}/n_p} \tag{2.3}$$

Bu denklem **Amdahl kanunu** olarak bilinir ve çoğunlukla eşitsizlik olarak ifade edilir. Bu, hemen hemen tüm durumlar için paralel hesaplamada elde edilebilecek "en iyi" hızlanmadır. Gerçek $S(n_p)$ hızlanması bu niceliğe eşit veya bu nicelikten daha düşüktür.

Amdahl kanunu paralelleştirme yapılırken göz önüne alınan bir işi yok eder: Eğer kodun seri kısmı paralelleştirilebilen kısmından çok küçük değil ise o zaman biz kaç tane işlemci olduğundan veya iletişimimizin ne kadar hızlı olduğundan bağımsız şekilde önemli bir hızlanma kazanmayacağızdır. Amdahl kanunu çok fazla idealistiktir. Kodun paralelleştirilmesinden gelen ilave maliyetleri ihmal eder. Bu sebeble bu kanunun genelleştirilmesi gereklidir:

Paralel hızlanmanın daha adil ve detaylı bir tarifi en azından iki tane daha ilave süreyi göz önüne almalıdır:

T_s : Orijinal tek işlemcili seri süre.

T_{is} : İşlemciler arası iletişimler gibi işleri yaparken harcanan (ortalama) ilave seri süre.

T_p : Orijinal tek işlemcili paralelleştirilebilen süre.

T_{ip} : İşi kurmak ve başlatmak için herbir işlemcinin harcadığı (ortalama) ilave süre. Bu süre çoğunlukla önem arzeden boşta bekleme süresini de içerebilir.

Genel olarak T_{is}'ye katkıda bulunan en önemli bileşen paralel altgörevler arasındaki iletişim için gerekli süredir. Bu iletişim süresi her zaman mevcuttur. Daha karmaşık işlerde, herbir işlemci üzerinde geliştirilen kısmi sonuçlar hesaplamanın devamı için diğer tüm işlemcilere gönderilir. Bu doğrultuda T_{is} verilen bir hesabın hızlanmasında aşırı derecede önemli rol oynar.

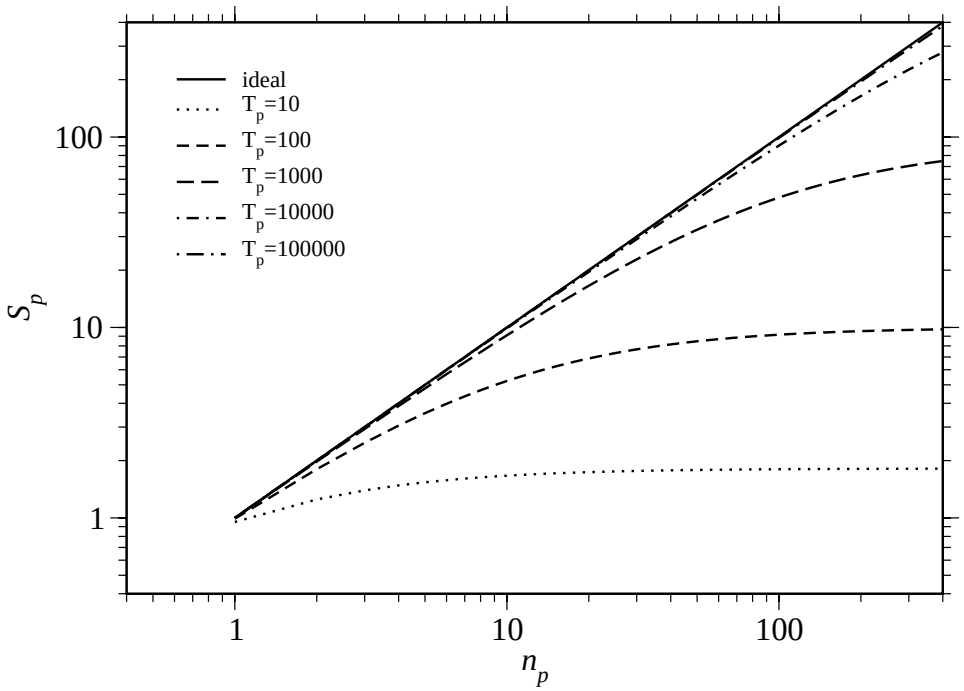
T_{ip}, n_p ve T_{is}'yi tek bir ifadede ($T_O(n_p)$ şeklinde "sabit giden süre") birleştirmek mümkündür; bu ifade işlemciler arası iletişimi, kurulumu, boşta (hazırda) beklemeyi ve paralel hesapta diğer sabit giden süreleri içine alır. Yukarıdaki tanım, gerçek zamanlı paralel işlerin çok daha fazla karmaşık olmasından dolayı hala *basitleştirilmiş* haldedir. Hatta bir çok durumda yukarıdaki tanım yetersiz kalabilmektedir.

Bu tanımları kullanarak ve biraz cebirsel işlem yaparak, belli bir işi n_p işlemci arasına bölmekten elde edilen paralel hızlanmaya ait daha gelişmiş bir saptama (fakat hala basit) aşağıdaki şekilde gelir:

$$S(n_p) = \frac{T_s + T_p}{T_s + n_p \times T_{is} + T_p/n_p + T_{ip}} \tag{2.4}$$

Bu ifade paralelleştirilebilen bir işin ölçekleme özellikleri hakkında en azından genel bir fikir elde edebilmek için yeterlidir. Boyutsuz nicelik hızlanmayı zamanın farklı göreli değerlerine karşı çizmek faydalıdır[12]. Aşağıdaki şekillerin hepsinde $T_s = 10$ ve T_s'ye kıyasla daha fazla işin paralelleştirilmesinin sistematik etkileri göstermek için T_p=10, 100,

1000, 10000, 100000 alınmıştır. Ölçekleme performasının ana belirleyicisi işlemciler üzerinde işleri kurmak için yapılması gereken seri iş miktarıdır, yani $T_{\rm is}$ ile temsil edilen süre. Tüm şekillerde $T_{\rm i} = 1$ şeklinde sabitlenmiştir.

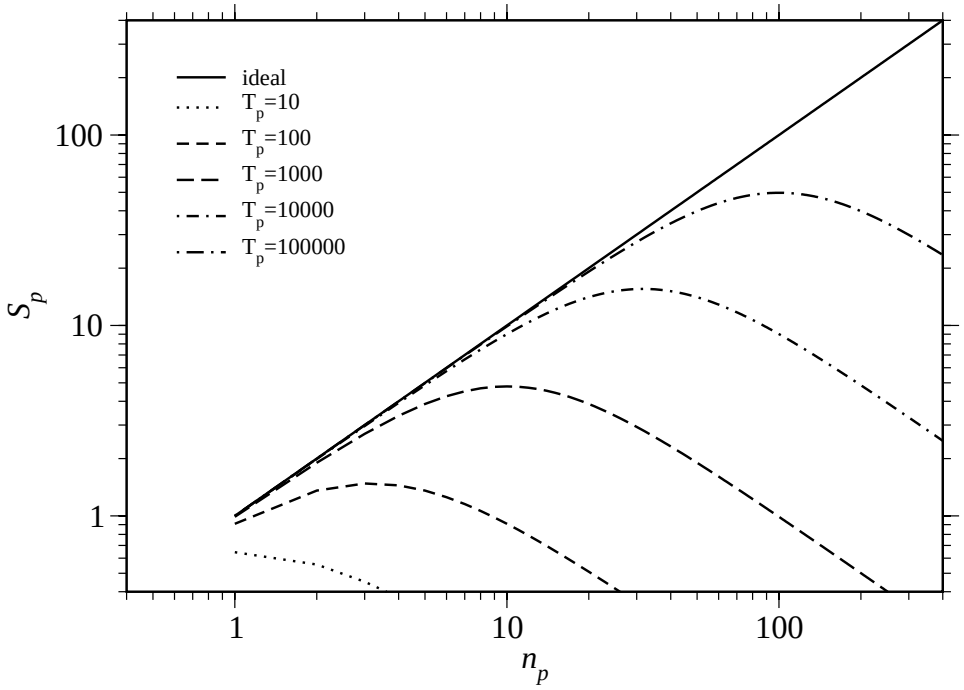


Şekil 2.17: $T_{\rm is} = 0$ ve $T_{\rm p}$=10, 100, 1000, 10000 ve 100000 için hızlanmalar.

Şekil 2.17 iletişim için geçen sürenin hesaplama için geçen süreye kıyasla ihmal edilebildiğinde olan bir tür ölçeklemeyi göstermektedir. Bu şekildeki eğriler kabaca Amdahl kanunundan beklenen eğrilerdir. Tüm şekillerdeki düz çizgi ile verilen eğrinin mükemmel lineer hızlanma olduğunu göz önüne almak gerekir. Bu mükemmel hızlanma küçük n_p veya büyük $T_{\rm p}$ değerlerinde bile hiç bir zaman elde edilememektedir.

Şekil 2.18'de $T_{\rm is} = 10$ olduğu gerçek paralel sistemler için nispeten daha tipik eğriler verilmektedir. Bu şekilde eğrilerin $T_{\rm p}$'nin $T_{\rm s}$ göre eğrilme avantajları gözükmektedir. Ayrıca herbir işlemcideki nispeten küçük seri iletişim süreçlerinin eğrilerin nasıl pik yapmalarına neden oldukları gözükmektedir. Bu pik noktaları geçildikten sonra işlemci ilave etmek hızlanmayı azaltmaktadır. $T_{\rm is}$'nin daha fazla artması eğrilerin daha erken ve daha küçük değerlerde pik yapmasına neden olur.

Şekil 2.19'de ise $T_{\rm is} \propto n_p^2$ şeklinde olup yüksek n_p değerlerinde seri işlemciler arası iletişimde harcanan süre çok uzun olmaktadır (yani örneğin herbir işlemci diğer tüm iş-

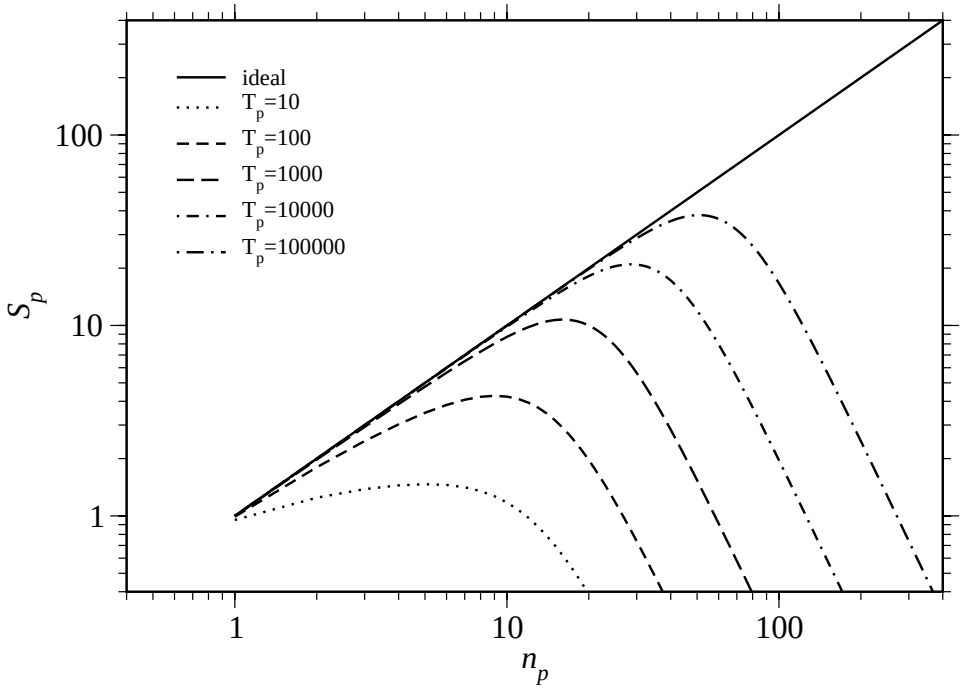


Şekil 2.18: $T_{\mathrm{is}} = 10$ ve T_{p}=10, 100, 1000, 10000 ve 100000 için hızlanmalar.

lemciler ile konuşmak zorundadır) ve bu durum uygun bir algoritma ile idare edilememektedir. n_p'e lineer olmayan bağımlılığın olduğu başka durumlar da vardır. Bu tür durumlarda şekilde görüldüğü üzere hızlanmanın işlemci başına ölçeklenebilirliliğinde önemli bir etki olmaktadır.

Özetlemek gerekirse, paralel performans öncelikle T_{s}, T_{p} ve T_{is} gibi belli nispeten basit parametrelere bağlıdır (detayda çok farklı unsurlar olmasına rağmen). Bu parametreler programlama kararlarında ve donanım tasarım kararlarında en azından kısmen bizim kontrolümüzdedirler. Ne yazık ki bu parametreler programcı tarafından erişilemeyen sistem ve ağ performansının birçok mikroskobik bileşenlerine bağımlıdır. Açıkça T_{p} bir bilgisayarın "hızı"na bağlı olacaktır, fakat tek bir bilgisayarın hızı işlemcisinin hızına lineer olmayan şekilde bağlı olabilir; önbelleğin büyüklüğü ve yapısı, işletim sistemi ve başka faktörler de etkili olacaktır.

Lineer olmayan karmaşıklıktan dolayı beklenen performansa dair bir saptama önceden yapılamaz. Mikroskobik ölçekte sistem performansına ait bir dizi nicel ölçümlerin ve ayrıca programda karşılaşılabilecek darboğazlar hakkında yaklaşımların elde olmasında yine de büyük fayda vardır.



Şekil 2.19: $T_{\text{is}} \propto n_p^2$ ve T_{p}=10, 100, 1000, 10000 ve 100000 için hızlanmalar ($n_p = 2$ için $T_{\text{is}} \approx 0$ ve $n_p = 100$ için $T_{\text{is}} \approx 50$).

Paralel algoritmaların performansını ölçmede diğer bir nicelik paralel hesabın verimidir. Yürütme zamanı problemin büyüklüğü ile değişmeye meyilli olduğu için, algoritmanın farklı problem büyüklüklerindeki performansını kıyaslarken yürütme zamanı normalize edilmelidir.

Amdahl kanunu yani işlemcilerin faydalı iş yaptıkları zamanın oranı paralel algoritmanın kalitesini ölçmede bazen daha uygun yoldur. Bu nicelik, algoritmanın paralel bir bilgisayardaki hesaplama kaynaklarını problemin büyüklüğünden bağımsız olarak kullanmasını karakterize eder. Verimi

$$\varepsilon = \frac{T(1)}{n_p T(n_p)} \tag{2.5}$$

şeklinde tanımlarız[9, 13, 14].

Bölüm 3

Mesaj Geçme Arayüzü (MPI)

Prensipte seri bir algoritma seri paradigmayı destekleyen herhangi bir mimariye taşınabilir. Bununla beraber programcılar bir program parçası şeklinde olan algoritmalarının taşınmasını isterler. Buna *"kaynak kod taşınabilirliği"* denir.

MPI'ın altında yatan ana fikir ve mesaj geçme programları için yukarıdaki ifadeler aynen doğrudur. MPI arayüzü C ve Fortran dillerinde yazılmış mesaj geçme programları bir çok çeşitli mimari arasında kaynak kod taşınabilirliğini sağlamaktadır[15].

Süreç iletişiminin temel fikri olan birinden diğerine mesaj gönderilmesi mantığının bulunması yıllar almıştır. Böylece günümüz mesaj geçme sistemleri kaynak kod taşınabilirliğini mümkün kılmaktadırlar[16].

MPI paralel işleme toplulukları arasında bir mesaj geçme standartı üretmek için ilk deneme olmuştur. Kırk farklı organizasyonu temsil eden altmış kişi (bunlar arasında Amerika ve Avrupa kıtasından paralel sistem üreticileri ve kullanıcıları da bulunmaktadır) bir araya gelerek **MPI Forumu** oluşturmuşlardır. Forumdaki tartışmalar, tüm topluluğa açık olmuş ve mesaj geçme sistemlerinin tasarımı ve kullanımı konusunda derin tecrübeleri olan bir çalışma grubu tarafından yönetilmiştir. Bu paralel sistemler arasında PVM, PARMACS gibi sistemler bulunmaktadır. İki yıllık süreçte amaçların belirtilmesi, görüşmeler ve incelemeler sonucunda **Mesaj Geçme Arayüzü (MPI)** standartını tanımlayan dökümantasyon tamamlanmıştır[5].

3.1 MPI tarihçesi

1992'deki bir atölye çalışmasıyla başlayan MPI Forum *SuperComputing'92* toplantısında organize edildi. MPI, paralel hesaplama topluluğunun geniş bir spektrumunun çabasıyla başarılı olmuştur. Üreticiler en iyi teknik insanlarını yollamışlardır. Taşınabilir kütüphanelerin yazarları ve uygulama programcıları en iyiyi üretmek için bir araya geldiler. MPI

Forum Ocak 1993'den başlayarak her altı ayda bir toplanmak kaydıyla 1994'ün yaz aylarında MPI'ı piyasaya sürdü[16].

MPI'ın modern tarihi 1995 yılının baharında başladı. Bu sırada Forum görüşme çizelgesini hazırlarken hem MPI-1 konusunda yetkin, hem de eşit sayıda yeni katılımcıyı kabul etmişti. Geçen üç yıl boyunca paralel hesaplama konusunda birçok değişiklikler olmuş ve görüşmeleri kapsayan (MPI-2 Forumu) iki yıl boyunca bu değişimler artarak süreci hızlandırmıştır[17].

Donanım tarafında ise MPP üreticilerinin bir topluluğu oluşmuştu[18]. Bunlar arasında Thinking Machines Corporation, Meiko ve Intel sayılabilir. Yeni girişler olmuş; Convex gibi (şimdi Hewlett-Packard firmasına aittir) ve SGI gibi (şimdi Cray Research firmasına aittir) MPI'ı destekleyen (paylaşımlı bellek modeli kullanan mesaj geçme modeli olarak) paralel hesaplamanın bir paylaşımlı bellek modelinde başarılı olmuş firmaların katılımı artmıştır[19]. Ayrıca birçok uygulama, hala paylaşımlı-bellek modelinin tepe performansını yakalayabilmek için NUMA makineleri mesaj geçme modelini kullanmaktaydı. Küçük ölçekli paylaşımlı-bellek çok işlemcili makineleri hemen hemen her PC ve işistasyonu üreticisi tarafından sağlanabilir hale gelmiştir[20].

3.2 MPI'ın kapsamı ve amaçları

MPI'ın ana amaçları kaynak kod taşınabilirliğini sağlamak ve çeşitli mimariler arasında etkin uyarlamayı mümkün kılmaktır. MPI heterojen paralel mimariler için destek sağlamakta ve pek çok fonksiyonelliği kullanıcıya/programcıya sunmaktadır. MPI kendi sahası dışında bilinçli olarak süreçlerin işlemcilere ilk kez yüklenilmesi, yürütme esnasında süreçlerin olaya dahil edilmesi, hata ayıklama, paralel Giriş/Çıkış (I/O) gibi konulara kesin destek vermektedir[21].

3.3 MPI ve uyarlamaları

MPI bir kütüphanedir. Bir MPI süreci, MPI fonksiyonları çağırarak diğer MPI süreçleri ile iletişime geçen bir C veya Fortran programından oluşur. MPI fonksiyonları programcıya farklı bir çok çeşit platformlar arasında uygun bir arayüzü sağlamaktadır.

LAM/MPI (Yerel Alan Çoklu Bilgisayarı) Amerikan Indiana Üniversitesi Açık Sistemler Laboratuarında araştırma, geliştirme ve bakımı yapılan yüksek başarımlı, ücretsiz dağıtılan MPI'ın açık kaynaklı bir uyarlamasıdır. LAM/MPI, MPI-1'in tamamını ve MPI-2'nin büyük bir bölümünü kapsar. LAM/MPI hakkında bilgi ve dökümanlara kendi resmi web sitesinden ulaşılabilir. Fakat bunların haricinde birkaç hata ayıklama ve iz-

leme yazılımını da içermektedir. *SSI (Sistem Servisleri Arayüzü)*, LAM/MPI'ın çekirdeğini oluşturan bir bileşen çatısı oluşturur. Bir çok geri-plan servisi SSI ile yapılmaktadır. SSI parametreleri komut satırından verilebilmekte, çalışma zamanında çevre değişkenleriyle kullanılabilmektedir[22].

RPI (İstek İlerleme Arayüzü), MPI'daki noktadan-noktaya mesajlaşma ilerleyişinde olduğu gibi bir kaynak süreçten hedef sürece kadar olan aşamaların LAM/MPI'da benzeri olan bir SSI modülüdür. *Boot Şeması*, LAM'da çalıştırılacak olan bir çoklu bilgisayarın tanımıdır. Bu genel çoklu bilgisayara dahil olan makinelerin isimlerinin olduğu bir listedir ve bu listedeki makineler sırayla LAM tarafından boot edilerek paralel çevreye dahil edilirler. *Boot*'un burdaki anlamı bir paralel çevrenin oluşturulması sırasında düğüm makinelerin bu çevreye eklenerek, paralel biçimde çalışır hale getirilmesidir.

Uygulama şeması, çoklu bilgisayardaki her düğüm makinede çalıştırılacak uygulamaların listesini içerir. LAM/MPI bir çok Beowulf kümesi'nde favori bir araçtır. Genelde LAM için "küme dostu" ifadesi kullanılmaktadır, çünkü küçük boyutlu artalan süreçleri yardımıyla hızlı süreç kontrolleri sağlaması ve uygulama başlatıp/bitirmedeki performansı buna neden olmuştur. LAM'ın 2006 yılı ikinci çeyreğinde kullanımda olan 7.1.2 sürümü aşağıdaki işletim sistemlerinde sorunsuzca çalışmaktadır.

- AIX, IRIX 6.5, Linux 2.4.x ve 2.6.x, Microsoft Windows (Cygwin sayesinde), MAC OS X 10.3, OpenBSD 3.5, Solaris 8 ve 9 işletim sistemleri.

LAM/MPI ortamında MPI paralel kodunun derlenmesi ve çalıştırılması Ek B'de verilmektedir.

MPICH (MPI Chameleon) [23] MPI'ın taşınabilir bedava bir dağıtımıdır. **MPICH2** tüm yeni MPI uyarlamalarını içerir şekilde, MPI-1 ve MPI-2'nin fonksiyonellikleri alınarak oluşturulmuş MPICH'in yeni versiyonudur. Bu versiyonda daha da detaylandırılmış bir MPI-IO (Girdi-Çıktı sistemi) geliştirilmiştir. Hatta Jumpshots (MPI grafikleri alınması işlemi) yani MPE (MP Envoironment kısaca Grafik Arayüzü) kullanımında bir seçenek vardır ve bu JDK 1.1 (Java Development Kit) ve JDK 1.2 ile kullanılabilmektedir[24].

3.4 Temel MPI fonksiyonları

Birçok ilk nesil ticari paralel bilgisayar, paylaşımlı adres uzayı mimarilerine bağlı olarak düşük maliyeti yüzünden mesaj geçme mimarisi tabanlıydı. Bu makineler için mesaj geçme doğal programlama paradigması olduğundan olay bir çok farklı mesaj geçme kütüphanelerinin üretilmesi ile sonuçlandı. Doğal olarak mesaj geçme *derleyici dili*'nin modern çağdaki versiyonu haline gelmiş oldu. Burada her donanım üreticisi kendi kütüphanesini sağlamakta, bu kütüphane kendi donanımında çok iyi çalışmakta, fakat diğer

üreticiler tarafından geliştirilen paralel bilgisayarlar ile uyumsuz olmaktaydı[25]. Çeşitli üreticiye-özgül mesaj geçme kütüphaneleri arasındaki birçok fark sadece söz dizimseldi. Bununla beraber sıklıkla bir mesaj geçme programını bir kütüphaneden bir diğerine taşıyabilmek için bazı ciddi mantıksal farklılıklardan dolayı muazzam derecede tekrar kodlama yapmak gerekiyordu[26].

Mesaj-Geçme Arayüzü MPI olarak bilinir ve esasen yukarıda anlatılan bu problemi çözmek üzere yaratılmıştır. MPI standart bir mesaj geçme kütüphanesi tanımlayarak bunun C veya Fortran dillerinin herhangi birisi kullanılarak oluşturulacak taşınabilir mesaj geçme programları üretilmesinde kullanılmasını sağlamıştır. MPI standartı her iki dilin söz dizimlerinde de olabildiğince iyi şekilde mesaj geçme programları yazımında çok kullanışlı olan çekirdek kütüphane fonksiyonları kümesinin mantığına dayanan söz dizimleri tanımlanmıştır (MPI fonksiyonları)[27]. MPI kütüphanesi 125'den fazla fonksiyon içermektedir, fakat ana fikri temsil eden kısım çok küçüktür. Bunun sonucu olarak, sadece altı tane temel MPI fonksiyonu ile (Tablo 3.1'de görüldüğü gibi) tamamen fonksiyonel mesaj geçme programları yazılabilir. Bu fonksiyonlar, MPI kütüphanesini başlatmak ve bitirmek, paralel hesaplama çevresi hakkında bilgi toparlamak, mesajları gönderip alma'yı sağlar[28].

MPI fonksiyon formatına bakacak olursak, bir MPI programında bir MPI fonksiyonu aşağıdaki gibi çağrılır:

- C dilinde (bundan sonra sadece *C* diye geçecek):

```
error= MPI_Xxxxx(parametre,......);
MPI_Xxxxx(parametre,......);
```

- Fortran dilinde (bundan sonra sadece *Fortran* diye geçecek):

```
CALL MPI_XXXXX(parametre,.........,IERROR)
```

MPI, C veya Fortran dilinde yazılmış programlara eklenmek istenildiğinde başlık (header) dosyası için:

C:

```
#include <mpi.h>
```

Fortan:

```
include 'mpif.h'
```

satırlarını programa eklemek gerekmektedir.

Genel bir basit MPI programının **kaba kodu (pseudo-code)** şu şekildedir:

```
include <mpi_baslik_dosyasi>    //Baslik dosyasini icerir.
<veri tipi> degisken            // Degiskenleri tanimlama
MPI_Init()                      // MPI'i baslat
                        ....

          Program govdesi olan kod kismi

                        ....
MPI_Finalize()                  // MPI'i sonlandir
```

MPI_Init	MPI başlatır.
MPI_Finalize	MPI bitirir.
MPI_Comm_size	Kaç adet süreç olduğunu tanımlar.
MPI_Comm_rank	Çağrılan sürecin etiketini tanımlar.
MPI_Send	Bir mesaj gönderir.
MPI_Recv	Bir mesaj alır.

Tablo 3.1: Temel MPI fonksiyonlarının işlevleri.

3.4.1 MPI'ın başlatılması ve bitirilmesi

MPI_Init diğer MPI fonksiyonları için herhangi bir çağrıyı ilklendiren fonksiyon olarak adlandırılır [29]. Amacı MPI çevresini başlatmaktır. MPI_Init'in programın çalıştırılması boyunca birden çok kereler çağrılması hataya sebep olur. MPI_Finalize, hesaplamanın sonu olarak adlandırılır. Öyle ki MPI çevresinin sonlandırılması için çeşitli temizleme görevlerini yerine getirir. MPI_Finalize çağrıldıktan sonra hiç bir MPI çağrısı icra edilmez, hatta MPI_Init bile icra edilmez. MPI_Init ve MPI_Finalize'ın her ikisi de bütün süreçler tarafından çağrılmalıdır, aksi durumda MPI'ın davranışı tanımlanamaz. Bu iki çağrının tam olarak fonksiyon prototipleri aşağıdaki gibidir:

C:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

Fortan:

```
INTEGER  IERROR
```

```
CALL MPI_INIT(IERROR)
CALL MPI_FINALIZE(IERROR)
```

Burada C dilindeki `MPI_Init` fonksiyon prototipi'nin (çağrı sıralanımı) argümanları olan *argc* ve *argv*, C programının komut satırı argümanlarıdır. Bir MPI uygulamasının programa geri dönmeden evvel uygulama tarafından işlenmiş olacak herhangi bir komut satırı argümanını *argc*'ye göre azaltmak için *argv* dizisinden silmesi beklenir. Böylece komut satırı işleme sadece `MPI_Init` çağrıldıktan sonra icra edilebilir. Başarılı bir yürütmenin ardından `MPI_Init` ve `MPI_Finalize` geriye `MPI_Success` kodunu döndürürler, aksi halde uygulamanın tanımladığı bir hata kodunu döndürürler.

Bu iki fonksiyonun bağlantıları ve çağrı sıralanımları, MPI tarafından izlenilen argüman ilişki mantığı ve isimlendirmede pratik gösterimini içerir. Tüm MPI fonksiyonları, veri tipleri ve sabitleri "`MPI_`" ile önek alarak oluşturulurlar. Başarılı bir tamamlamanın geri dönüş kodu `MPI_Success`'dir. Bu ve benzeri MPI sabit ve veri yapıları C veya Fortran için **"mpi.h"** veya **"mpif.h"** dosyası (başlık dosyası) ile her MPI programına eklenerek kullanılır[29]. Bu başlık dosyası tüm bu bahsedilen veri yapılarını içerir.

3.4.2 İletişimciler ve bilgileri

MPI'yı kullanmanın temel ana fikirlerinden biri *iletişim domeni*'dir. Bu domen (*alan veya evlek de denilebilir*) genel olarak, süreçlerin birbirleri ile iletişiminin mümkün kılındığı bir süreç kümesidir. İletişim domeni hakkındaki bilgi `MPI_Comm` tipli değişkenlerde saklanır ve bunlara **iletişimci** adı verilir.

Bu iletişimciler, bütün mesaj transferi yapan MPI fonksiyonları için argüman olarak ve mesaj transferi işlemine katılan süreçlerin tek tek tanımlanmasında kullanılmaktadır. Şunu da göz ardı etmemek gerekir ki her bir süreç bir çok farklı (*örtüşüm olabilir*) iletişim domen'ine ait olabilir.

İletişimciler, birbirleri ile iletişim yapabilen süreçlerin bir kümesini tanımlamada kullanılırlar. Bu süreçlerin kümesi bir iletişim domeni'ni biçimlendirir. Genelde tüm süreçler birbirleri ile iletişime ihtiyaç duyarlar. Bu sebepten MPI varsayılan bir iletişimci tanımlar, buna `MPI_COMM_WORLD` adı verilir. Bu iletişimci paralel yürütmeye karışan tüm süreçleri içerir. Bununla beraber çoğu durumda sadece süreç grupları arasında (*örtüşüm olabilir*) iletişim icra edilmek istenir. Her bir grup için farklı bir iletişimci kullanarak bir mesajın başka gruptakiyle karışmaması sağlanır[30].

`MPI_Comm_size` ve `MPI_Comm_rank` fonksiyonları aslen kaç tane süreç olduğunu tanımlama ve çağrılan süreçleri etiketleme amacıyla kullanılmaktadırlar. Bu fonksiyonlara ait fonksiyon prototipleri aşağıdaki gibidir:

C :

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran :

```
INTEGER COMM, SIZE, IERROR
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
```

`MPI_Comm_size` fonksiyonu, *comm* iletişimcisine ait olan süreç sayısı *size* değişkenini geri döndürür. Bir iletişimciye ait her süreç bir tek olarak *rank*'ı (sırası) tarafından tanımlanır. Bir sürecin sırası, sıfırdan iletişimcinin bir eksiği büyüklüğe kadarlık bir aralıktaki tamsayıdır (yani burada *rank*, sıfır ile *comm-1* arasında bir pozitif tamsayı olarak gösterilebilir). Bir süreç, iki argüman alan `MPI_Comm_rank` fonksiyonu kullanılarak bir iletişimci içindeki sırası ile tanımlanır. Bu iki argüman sırasıyla; bir iletişimci ve bir tamsayı *rank* değişkenidir. Konuya geri dönecek olursak *rank* değişkeni sürecin sırasını saklar. Şunu unutmamak gerekir ki her süreç bu fonksiyonların herhangi birisinden çağrılırsa sağlandığı iletişimciye ait olmak zorundadır aksi halde hata oluşur[30].

3.4.3 MPI'ın bitmeden durdurulması

Bir MPI programında, *comm* grubundaki tüm işlemcilerin çalışmanın herhangi bir yerinde durdurulması istenirse `MPI_Abort` fonksiyonu çağrılır. Bu fonksiyonun genel şekli şöyledir:

C :

```
int MPI_Abort(MPI_Comm comm)
```

Fortran :

```
INTEGER COMM, IERROR
CALL MPI_ABORT(COMM, IERROR)
```

3.4.4 MPI veri tipleri

MPI veritipleri ve bunların C dilindeki karşılıkları Tablo 3.2'de ve Fortran dilindeki karşılıkları Tablo 3.3'de verilmektedir. Tüm C veya Fortran veri tiplerine karşılık gelecek MPI veritipleri var olması konusunda bir kural yoktur. Bu dillerde olmayan MPI veritiplerine iki ek tip `MPI_BYTE` ve `MPI_PACKED` olarak belirtilmiştir.

MPI veritipi	C dili veritipi
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tablo 3.2: MPI tarafından ve C dili tarafından sağlanan veritipleri arasındaki ilişki.

MPI veritipi	Fortran dili veritipi
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Tablo 3.3: MPI tarafından ve Fortran dili tarafından sağlanan veritipleri arasındaki ilişki.

3.4.5 Basit bir MPI programı

Şimdi bu dört MPI fonksiyonunu kullanarak **"Merhaba Dünya"** yazısını her düğüme mesaj olarak geçen bir programı C dilinde yazarsak:

```
Program:
1    #include <mpi.h>
2
3    main(int argc, char *argv[])
4    {
5       int islemci_sayisi, siram;
6
7       MPI_Init(&argc, &argv);
8       MPI_Comm_size(MPI_COMM_WORLD, &islemci_sayisi);
9       MPI_Comm_rank(MPI_COMM_WORLD, &siram);
10      printf("%d adet surecten %d inci, Merhaba Dunya dedi.\n",
11                   islemci_sayisi, siram);
12      MPI_Finalize();
13    }
```

şeklinde bir program olur. Bu programın ekran çıktısına bakacak olursak:

```
mpirun -np 4 a.out
4 adet surecten 0 inci, Merhaba Dunya dedi.
4 adet surecten 3 inci, Merhaba Dunya dedi.
4 adet surecten 1 inci, Merhaba Dunya dedi.
4 adet surecten 2 inci, Merhaba Dunya dedi.
```

burada görüldüğü gibi karışık sırada süreçlere erişim olmuş ve olaya dahil olan süreç düzgün olarak cevap vermiştir.

3.5 Noktadan-noktaya iletişim

Noktadan-noktaya iletişim daima iki süreci içerir. Süreçlerden biri diğerine mesajı gönderir. Bu durum ileride anlatılacak olan MPI'daki toplu iletişimden farklılık gösterir.

Bir mesaj göndermek için kaynak süreç, uygun iletişimci grubundaki sürecin sırası cinsinden hedefi belirten MPI çağrısını icra eder. Hedef süreçte bu mesajı almak için MPI çağrısı yapmak zorundadır[31].

3.5.1 MPI_Send ve MPI_Recv fonksiyonları

MPI'da mesajları göndermek ve almak için temel fonksiyonlar esasen, `MPI_Send` ve `MPI_Recv` fonksiyonlarıdır. Bu fonksiyonların çağrı sıralanımları aşağıdaki gibidir:

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag,MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag,MPI_Comm comm, MPI_Status *status)
```

Fortran :

```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
               IERROR)
TYPE BUF(*)
INTEGER COUNT, DATATYPE, DEST, SOURCE, TAG, COMM
INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

`MPI_Send` tarafından gönderilmiş mesajın hedefi tek olarak *dest* ve *comm* argümanları tarafından belirlenir. Bunlardan *dest* argümanı, *comm* iletişimcisi tarafından belirlenmiş iletişim domeni'ndeki hedef sürecin sırasıdır. Her mesaj kendisiyle ilişkilendirilmiş tamsayı değerli bir *tag* etiketine sahiptir. Bu farklı tipteki mesajları ayırt etmekte kullanılır. Mesaj etiketi *tag* sıfırdan MPI tarafından tanımlanan `MPI_TAG_UB` sabitine kadar olan aralıktaki değerleri alabilir. `MPI_TAG_UB` değeri uygulamaya-özgül olmakla beraber, en fazla 32.767 olur[32].

`MPI_Recv` *comm* argümanı ile tanımlanan iletişim domeni'nde *source* argümanıyla verilen kaynağın rank'ı ile belirtilen bir süreç tarafından gönderilmiş bir mesajı alır. Gönderilen mesajın etiketi *tag* argümanı ile belirlenmiş olmalıdır. Eğer aynı süreçten özdeş etiketli bir çok mesaj varsa, o zaman bu mesajların herhangi biri alınır. MPI, *source* ve *tag* gibi her iki argümanın özelliklerini tanımlar. Eğer *source* `MPI_ANY_SOURCE` sabiti olarak verilirse, iletişim domeni'ndeki herhangi bir süreç mesajın kaynağı olabilir. Benzer olarak eğer *tag* etiketi `MPI_ANY_TAG` olarak verilirse, herhangi bir etikete sahip olan mesajlar kabul edilir.

Bir mesaj alındıktan sonra, *status* değişkeni `MPI_Recv` işlemi hakkında bilgiyi almak için kullanılır. C dilinde *status*, `MPI_Status` veri yapısı kullanılarak saklanır.

3.6 Toplu iletişim

MPI verileri dağıtma, toplama, global toplamlar yapma vb. gibi işler için çok çeşitli fonksiyonlar sağlar. Bu tür fonksiyonlar, toplu işlemler yapıldığından dolayı "toplu iletişim" fonksiyonları olarak ifade edilirler. Toplu iletişimi noktadan-noktaya iletişimden ayıran temel husus, toplu iletişimde belirtilen iletişimcideki her bir sürecin işleme dahil olmasıdır.

MPI tarafından sağlanan tüm toplu iletişim fonksiyonları, topluca işleme katılan süreçlerin bir grubunu tanımlayan bir iletişimciyi argüman olarak alırlar. İşleme katılan bu

iletişimciye ait süreçlerin hepsi toplu iletişim fonksiyonunu çağırmak zorundadır. Durum böyle olsa da toplu iletişim işlemleri bir bariyer varmış gibi davranmaz, daha çok son durum senkronizasyonu gibi davranırlar. Eğer global senkronizasyon ortak çağrıdan önce veya sonra icra ediliyorsa, doğru şekilde davranacak paralel programların yazılması zorunludur. Çünkü sanal olarak senkronize olmuş işlemler etikete ihtiyaç duymaz[32].

3.6.1 Barrier (engel koyma)

Bariyer (Engel) senkronizasyon işlemi, MPI'da `MPI_Barrier` fonksiyonu kullanılarak yapılır:

C:

```
int MPI_Barrier(MPI_Comm comm)
```

Fortran:

```
CALL MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

`MPI_Barrier` için tek argüman senkronize olması istenilen süreçlerin grubudur. `MPI_Barrier` çağrısı sadece gruptaki tüm süreçler bu fonksiyonu çağırdıktan sonra geri döner (sonlanır).

3.6.2 Broadcast (yayımlama)

Süreçlere gönderilecek veri için "Birisinden-Herkese" (One-to-All) yayımlama işlemi, MPI'da `MPI_Bcast` fonksiyonu ile yapılır.

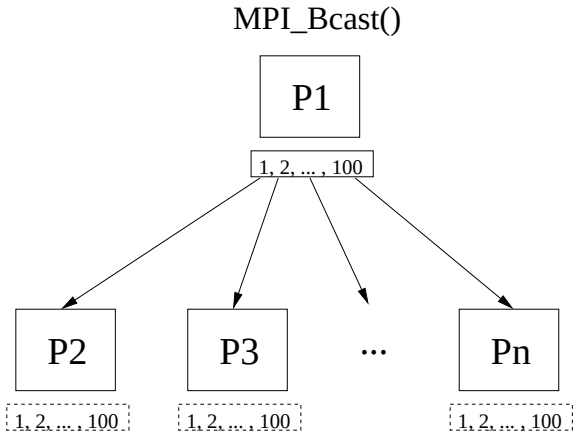
C:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int source, MPI_Comm comm)
```

Fortran:

```
CALL MPI_BCAST(BUF, COUNT, DATATYPE, SOURCE, COMM, IERROR)
TYPE BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, COMM, IERROR
```

Burada `MPI_Bcast` *source* sürecinin *buf* tamponunda saklanan veriyi gruptaki tüm süreçlere gönderir. Her süreç tarafından alınan veri *buf* tamponunda saklanır. Veri *datatype* veri tipindeki *count* adetince girişten oluşmuştur[33]. Ayrıca **source** sürecinden gönderilen veri miktarı, her süreç tarafından alınan veri miktarına eşit olmak zorundadır, yani *count* ve *datatype* alanları tüm süreçlerde birbirlerine uygun olmalıdır. Bu işlemin şeması Şekil 3.1'de verilmektedir.



Şekil 3.1: Bir düğümden diğer düğümlere yayımlama.

3.6.3 Reduction (indirgeme)

Süreçlerden elde edilen veri için Herkesden-Birisine indirgeme işlemi, MPI'da `MPI_Reduce` fonksiyonu ile yapılır.Bu işlem Şekil 3.2'de açıkça görülmektedir.

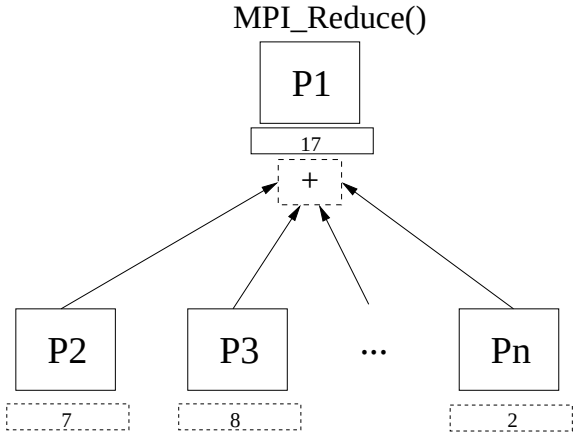
C :

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
                           MPI_Datatype datatype, MPI_Op op,
                           int target, MPI_Comm comm)
```

Fortran :

```
CALL MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE,
                          OP, TARGET, COMM, IERROR)
TYPE SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, TARGET, COMM, IERROR
```

Burada `MPI_Reduce` gruptaki her sürecin *sendbuf* tamponundaki saklanmış elemanları *op* ile belirtilen işlem cinsini uygulayarak birleştirilir ve sonra *target* rank (sıra)'ına sahip sürecin *recvbuf* tamponuna değeri aktarır[33]. Bu durumda *sendbuf* (gönderim tamponu) ve *recvbuf* (alım tamponu) aynı büyüklüğe sahip olmalıdır. MPI, önceden tanımlı olan ve *sendbuf*'da saklanan elemanları birleştirmede kullanılan indirgeme işlemlerini sağlamaktadır. Bu işlemler Tablo 3.4'de görülmektedir.



Şekil 3.2: Düğümlerdeki verinin tek bir düğüm üzerindeki tek bir veri olarak indirgenmesi.

İşlem	Anlamı	Veritipleri
MPI_MAX	Maksimum	Tamsayı ve kayan nokta
MPI_MIN	Minimum	Tamsayı ve kayan nokta
MPI_SUM	Toplam	Tamsayı ve kayan nokta
MPI_PROD	Çarpım	Tamsayı ve kayan nokta
MPI_LAND	Mantıksal AND	Tamsayı
MPI_BAND	Bit-bazlı AND	Tamsayı ve byte
MPI_LOR	Mantıksal OR	Tamsayı
MPI_BOR	Bit-bazlı OR	Tamsayı ve byte
MPI_LXOR	Mantıksal XOR	Tamsayı
MPI_BXOR	Bit-bazlı XOR	Tamsayı ve byte
MPI_MAXLOC	Azami-Asgari değer-yerleşimi	Veri-Çiftleri
MPI_MINLOC	Asgari-Asgari değer-yerleşimi	Veri-Çiftleri

Tablo 3.4: Önceden tanımlı indirgeme işlemleri.

Eğer indirgeme işleminin sonucu tüm süreçler tarafından istenilirse MPI tarafından sağlanan `MPI_Allreduce` fonksiyonu kullanılır. `MPI_Allreduce` için çağrı sıralanımı aşağıdaki gibidir:

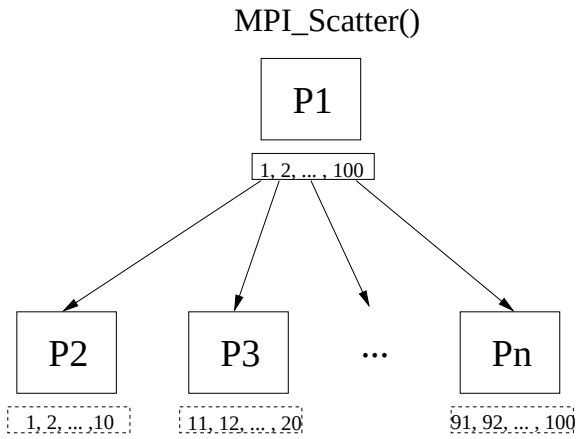
C:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
                          MPI_Datatype datatype,
                          MPI_Op op, MPI_Comm comm)
```

Fortran:

```
CALL MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE,
                           OP, COMM, IERROR)
TYPE SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

Ayrıca `MPI_Alltoall` fonksiyonu ile "Herkesten-Herkese" bir veri dağıtımı şekli de bulunmaktadır.



Şekil 3.3: Düğümdeki verinin diğer düğümler üzerine saçılması.

3.6.4 Scatter (saçılma) ve Gather (bir araya getirme)

MPI'ın bir diğer veri dağıtımı ve toparlanmasıyla alakalı komut seti çifti ise `MPI_Scatter` ve `MPI_Gather` olarak verilmektedir. Bunlardan `MPI_Gather`'ın `MPI_Reduce`'un sahip olduğu gibi `MPI_Allgather` benzeri genelleştirilmiş versiyonları bulunur[34]. Temel amaçları süreçler arasında veri dağıtımı ve sonra verinin bir noktada toparlanmasıdır.

Saçılma işlemi, MPI'da `MPI_Scatter` fonksiyonu ile yapılır. Bu fonksiyonun çağrı sıralanımı şöyledir:

C:

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype senddatatype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

Fortran:

```
CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDDATATYPE,
                 RECVBUF, RECVCOUNT, RECVDATATYPE,
                              SOURCE, COMM, IERROR)
TYPE SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDDATATYPE, RECVCOUNT
INTEGER RECVDATATYPE, SOURCE, COMM, IERROR
```

Burada *source* (kaynak) süreç kendisi de dahil olmak üzere, *sendbuf* gönderim tamponunun farklı bir parçasını her süreç için gönderir. Alınan veri, *recvbuf* alım tamponunda saklanır. Süreç *i* kaynak sürecin *sendbuf* tamponunun *i * sendcount* yerleşiminden başlayarak *senddatatype* veri tipli *sendcount* sürekli elemanlarını alır (*sendbuf, senddatatype* veri tipi ile aynı veri tipine sahiptir). `MPI_Scatter` fonksiyonu, *sendcount*, *senddatatype*, *recvcount*, *recvdatatype*, *source*, *comm* argümanları için aynı değerlere sahip süreçlerin tümü tarafından çağrılmak zorundadır.

MPI'da bir araya getirme işlemi, `MPI_Gather` ve `MPI_Allgather` ile yapılır:

C:

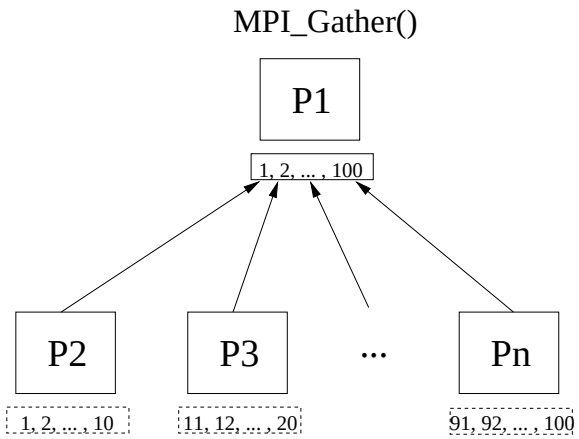
```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvdatatype, MPI_Comm comm)
```

Fortran:

```
CALL MPI_GATHER(SENDBUF, SENDCOUNT, SENDDATATYPE, RECVBUF,
                RECVCOUNT, RECVDATATYPE, TARGET, COMM,
                                                 IERROR)
CALL MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDDATATYPE, RECVBUF,
                   RECVCOUNT, RECVDATATYPE, COMM, IERROR)
TYPE SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDDATATYPE, RECVCOUNT
INTEGER RECVDATATYPE, TARGET, COMM, IERROR
```

`MPI_Scatter` ve `MPI_Gather` Şekil 3.3 ve 3.4'de temsil edilmektedir. Her süreç (burada *target* sürecide buna dahildir) *sendbuf* dizisinde saklanan veriyi *target* (hedef) sürece

gönderir. Bunun sonucu olarak *comm* iletişimcisindeki p tane süreç varsa, hedef süreç toplamda p tamponunu alır. `MPI_Allgather` ve `MPI_Gather`'da aynı koşullardaki argümanlar için gönderilen verinin, kaç adet olduğu ve tipi, *sendcount* ve *senddatatype* argümanlarıyla, alınan verininkiler ise, *recvcount* ve *recvdatatype* argümanları ile belirtilir. `MPI_Allgather`'ın `MPI_Gather`'daki argümanlardan sadece *target*'a sahip olmadığını göz önüne alarak; `MPI_Allgather`'ın sadece hedef süreçte değil, aksine tüm süreçlerde verinin bir araya getirildiğini ve bunun MPI'ın yapısından kaynaklandığını belirtmek gerekir. Ayrıca `MPI_Allgather`'da her süreç bir *recvbuf* (alım tamponu) tamponu dizisini (bir araya getirilmiş verinin saklanıldığı dizi) sağlamak zorundadır[35, 36].



Şekil 3.4: Düğümlerdeki verilerin tek bir düğüm üzerindeki tek bir veri olarak bir araya getirilmesi.

3.7 MPI_Wtime() ile zaman tutma

MPI programında zaman tutmak istenildiğinde `MPI_Wtime` fonksiyonu kullanılır. Bu fonksiyonun çağrı sıralanımı şöyledir:

C :

```
double MPI_Wtime(void)
```

Fortran :

```
CALL MPI_WTIME()
DOUBLE PRECISION
```

Bir MPI programında zaman tutmayı aşağıdaki biçimde yapabiliriz:

```
Başla
MPI Başlat
...
t1=MPI_Wtime()
...
Hesaplama işlemleri ve geçen süre boyunca yapılan işlemler kısmı...
...
t2=MPI_Wtime()
...
print (‘‘Gecen sure=’’,(t2-t1))
...
MPI bitir
Dur.
```

3.8 Paralel hesaplamaya bir örnek

Bu kısımda MPI kütüphanesi kullanılarak yapılan paralel hesaplamaya bir örnek yapalım. Seçtiğimiz örnek nümerik analizden iyi bildiğimiz belirli integralin nümerik hesabı olsun. Yani bir $f(x)$ fonksiyonunun α'dan β'ya kadar olan $\int_{\alpha}^{\beta} f(x)dx$ integralinin hesabını paralel hesap tekniğini kullanarak yapmak isteyelim:

Trapez kuralına göre, Şekil 3.5'de gösterildiği gibi α'dan β'ya kadar olan x aralığı h uzunluklu toplam n eşit parçaya bölünür. $f(x)$ eğrisi altında kalan alana

$$A = \sum_{i=1}^{n} a_i = \frac{h}{2} \sum_{i=1}^{n} (f_i + f_{i+1}) \tag{3.1}$$

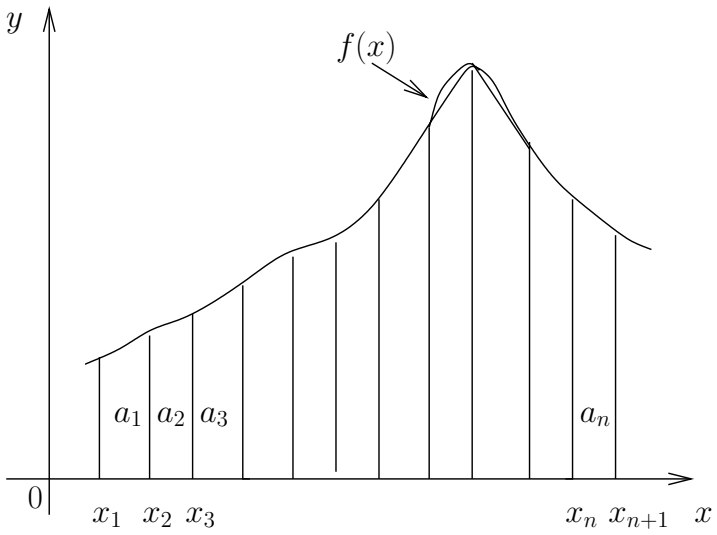
şeklinde yaklaşılabilir. Bu ifadede uç noktalar yani $i = 1$ ve $i = n + 1$ hariç diğer tüm noktalarda $f(x)$ değeri iki defa gözükür. Bu tarzda Trapez kuralını kullanarak integralin yaklaşı değeri şu şekilde yazılır [37]:

$$\int_{\alpha}^{\beta} f(x)dx \approx \frac{h}{2}\{f_1 + 2f_2 + \ldots + 2f_n + f_{n+1}\} = \frac{h}{2}\left\{f(\alpha) + f(\beta) + h\sum_{i=1}^{n-1} f(\alpha + ih)\right\} \tag{3.2}$$

Şimdi bizim $f(x)$ fonksiyonumuz $f(x) = x^2$ şeklinde bir fonksiyon olsun ve biz

$$\int_{1}^{5} f(x)dx$$

integralini hesap edelim



Şekil 3.5: Trapez kuralı integrasyon.

- Analitik olarak

$$\int_1^5 f(x)dx = \int_1^5 x^2 dx = \frac{x^3}{3}\mathrm{I}_1^5 = \frac{124}{3} \cong 41.33333$$

olarak elde edilir.

- Bu nümerik integrasyonun seri hesabına ait kaba kod aşağıdaki gibidir:

```
BEGIN
 a,b ve n degerlerini al;
 h=(b-a)/n;
 trapez kuralı ile hesabı yap;
END.
```

$n = 1000$ için bu hesap 41.33284 şeklinde elde edilir.

- Bu nümerik integrasyonun paralel hesabına ait kaba kod ise aşağıdaki gibi yazılabilir.

```
BEGIN
 MPI'ı baslat;
 a,b ve n degerlerini al;
```

```
 h=(b-a)/n;
 n adet alan hesabını p kadar islemciye paylastır,
 eger (rank=0) ise rank=1'den (p-1) rankına kadar
      olan islemcilerden sonucları al
      ve toplam sonucu hesap et,
 eger (rank>0) ise kendi hakkına düsen alanları
      trapez kuralı ile hesapla,
      sonucu rank=0'a bildir;
 MPI'ı bitir;
END.
```

$n = 1000$ ve $p = 10$ işlemci için bu hesabın sonucu 41.55432 şeklinde yaklaşık olarak bulunur.

Bölüm 4

Monte Carlo simulasyonunda paralel hesaplama

Simulasyon çalışmaları teorik ve deneysel araştırma alanlarının arasında giderek daha çok önem kazanan bir alan olmuştur. Gerçek deneylerin hem çok maliyetli olması hem de bazen tehlikeli ya da sakıncalı olması nedeni ile bilgisayar ortamında yapılan simulasyon çalışmaları deneylerin maliyetlerinin azaltılmasını sağlarken bazen de deneysel çalışmalar ile özdeş olmaktadır. Diğer yandan simulasyon vermiş olduğu sonuçlar ile teorik öngörülerin doğruluğunu ispatlayabilmektedir. Özellikle iklim çalışmalarında, biyolojik sistemler üzerine olan çalışmalarda, jeofizik ve jeolojik çalışmalarda simulasyon çok büyük önem kazanmıştır[38].

Karmaşık sistemlerin simulasyonu çoğunlukla çok uzun (aşırı) zaman boyunca programların koşturulmasını gerekli kılmaktadır (örneğin tipik bir iletim simulasyonunda yaklaşık 10 yıl CPU zamanı). Bu sebeble simulasyon tekniği ne olursa olsun paralel hesaplama bu zorluğa çözüm sunmaktadır.

Bu tez çalışmasında yukarıda anlatılan karmaşık sistemler için bir temel olabilecek bir sistemin Monte Carlo simulasyonunun paralel hesabı yapılmıştır. Bu sistem N tane parçacıktan oluşan “ideal polimer zinciri”dir. Bu zincirin davranışını inceleyen Monte Carlo simulasyonunun paralelleştirilmesi yapılmış, seri ve paralel hesaba dair sonuçlar hem kendi aralarında hem de teorik öngörü ile karşılaştırılmıştır. Paralel hesaba dair performans analizleri detaylıca verilmiştir.

Bu çalışmadaki program kodları C dilinde yazılmış ve Linux ortamında Intel tabanlı makinalarda (TÜBİTAK ULAKBİM’deki makinalarda ve Maltepe Üniversitesi’ndeki makinalarda) çalışmalar yapılmıştır. Paralel hesap MPI kütüphanesi kullanılarak yapılmıştır.

4.1 İdeal polimer zinciri

Polimer, kovalent kimyasal bağlarla bağlanmış tekrarlanan birimler ve yapısal birimlerden oluşan çok uzun bir molekülü tarif etmekte kullanılan bir terimdir. Terim iki kısımdan oluşan Yunanca kaynaklı bir terimdir. Poly, çok anlamına, Meros ise parça anlamına gelmektedir. Polimerleri diğer moleküllerden ayıran ana özelliği bir çok benzer, aynı veya birbirini tamamlayan zincirdeki moleküler altbirimlerin tekrarı olmasıdır. Bu alt birimler, monomer olarak adlandırılır ve polimerizasyon olarak bilinen kimyasal bir reaksiyon sırasında birbirlerine bağlanırlar[39]. Monomerler arasındaki farklılıklar esneklik ve güçlülük gibi bazı özelliklere etki eder. Proteinlerde, diğerlerine göre biyolojik-aktiflik şeklinde adapte olabilirlik bu farklılıklardan dolayı oluşur (kendiliğinden bir araya gelme). Benzer monomerler, reaktif olmayan gruptakiler bir polimer zincirinde bir parçanın zincire adapte olmasının sonucudur. Böyle bir durum ideal bir zincir matematik modeliyle tanımlanır. Bununla beraber bir çok polimer organiktir. Bunlar karbon temelli monomerlerdir. Ayrıca inorganik polimerlerde vardır, örneğin silikon ve oksijen atomlarının karışımıyla oluşmuş silikonlar buna örnektir.

İdeal polimer zinciri: Bir polimeri tanımlamada kullanılan en basit model bir ideal zincirdir. Bir polimeri sadece bir rasgele yürüme olarak kabul edersek, monomerler arasındaki etkileşimlerin her bir çeşitini yaklaşık olarak yansıtır. Bununla beraber, bu polimerlerin fiziğinin anlaşılmasında kolaylık sağlar[6].

Bu modelde, monomerler sabit l uzunluğunda katı çubuklarla bağlanmış olan noktasal cisimler olarak alınırlar ve komşu monomerlerin yerleşimleri ve uyumlarından tamamen bağımsız bir uyuma sahiptirler.

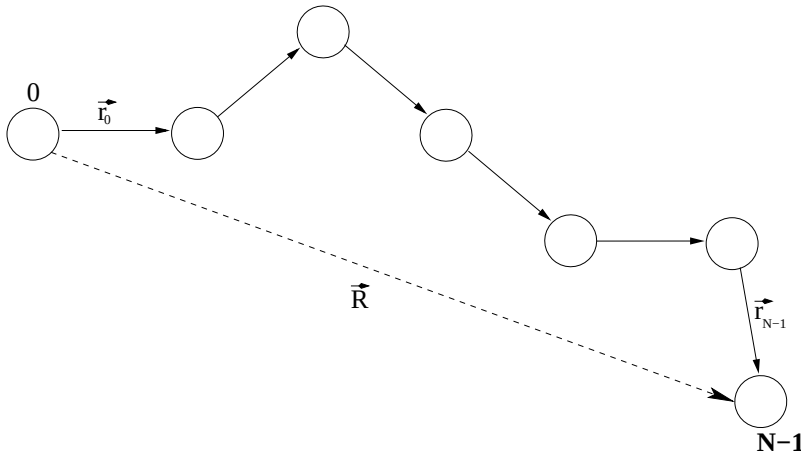
N tane monomerden oluşan polimer zinciri için $L = Nl$ azami açılma uzunluğudur. Bu en basit yaklaşımda, monomerler arasındaki etkileşimler yok sayılarak termodinamik denge anlamıyla alınan polimerin şeklinden bağımsız olarak enerjisi Maxwell-Boltzmann dağılımına göre belirlenir.

Burada bir ideal zincirdeki *uçtan uca* (end-to-end) toplam vektörü $\vec{R}$ ve kişisel olarak her bir monomerle alakalı $\vec{r}_0, ..., \vec{r}_{N-1}$ vektörlerini alalım. Bu rasgele vektörler uzayda, üç yöne sahiptir.

Burada, zincirde örtüşmeyen iki uç için;

$$\langle \vec{R} \rangle = \sum_{i=0}^{N-1} \langle \vec{r}_i \rangle = 0 \tag{4.1}$$

Ayrıca, buradaki $\langle\rangle$ braketleri, rasgele bir vektör veya rasgele bir değişken anlamındadır (alınan değer bölü zaman). Çünkü $\vec{r}_0, ..., \vec{r}_{N-1}$ birbirinden bağımsızdır. Merkezi limit teoremi $\vec{R}$, normal dağılıma göre (Gauss dağılımı) dağıtılır. Üç boyutta $R_x,\ R_y,\ R_z$ olarak



Şekil 4.1: İdeal Polimer Zinciri.

normal bir dağıtım yapılır.

$$\sigma^2 = \langle R_x^2 \rangle - \langle R_x \rangle^2 = \langle R_x^2 \rangle - 0 \tag{4.2}$$

$$\langle R_x^2 \rangle = \langle R_y^2 \rangle = \langle R_z^2 \rangle = N\frac{l^2}{3} \tag{4.3}$$

Böylece $\langle \vec{R}^2 \rangle = Nl^2 = Ll$ olur. Zincirin uçtan uca vektörü, aşağıdaki olasılık dağılımı fonksiyonu ile alakalı bir dağılımdır.

$$P(\vec{R}) = \left(\frac{3}{2\pi N l^2}\right)^{3/2} e^{-\frac{3\vec{R}^2}{2Nl^2}} \tag{4.4}$$

Polimerin ortalama uçtan uca mesafesi:

$$\sqrt{\langle \vec{R}^2 \rangle} = \sqrt{N}l = \sqrt{Ll} \tag{4.5}$$

olarak verilir.

Aynı zamanda bu basit sistem, dalgalanma genliği olan yukarıdaki uçtan uca ortalaması termodinamik limitteki açılmış polimer zincirinin uzunluğu Nl karşısında gözardı edilebilir. Bu sonuç istatistiksel sistemlerin genel özelliğidir.

Olasılık yoğunluğu ifadesi $P(\vec{R})$ yukarıda gösterildiği gibi doğrudan olmaz; öyleki merkezi limit teoremi'nin bir boyutlu uygulamasından R_x, R_y ve R_z'in ortalanmış dağılım değişkeni $\frac{Nl^2}{3}$'e göre dağıldığı algılanabilir. Böylece $P(\vec{R})$ için yukarıda verilen ifade, R_x, R_y ve R_z'in böyle bir dağılımıyla uyumlu tek seçenek değildir. Ancak, $\vec{r}_i, ..., \vec{r}_N$ vektörlerinin bileşenleri ilgilendiğimiz "rasgele yürüme" ile ilişkili olmadığı için R_x, R_y ve R_z'de

aynı şekilde bu rasgele yürüme ile ilişkili değillerdir[40]. Bu ek koşul sadece $\vec{R}$'in $P(\vec{R})$'ye göre dağılmasıyla sağlanabilir. Diğer yandan bu sonuç merkezi limit teoremi'nin çok boyutlu genelleme uygulamasıyla veya simetri argümanlarının aracılığıyla gösterilebilir.

Yukarıdakilere ilave olarak, ideal polimer zincirinin davranışının istatistik fizikteki rasgele yürüme hareketi olduğuna dikkat etmek gerekir[41].

Polimer zincirinin büyüklüğünü ölçmede alternatif bir nicelik deneysel olarak da ölçülebilen jirasyon yarıçapıdır. Jirasyon yarıçapı

$$R_g^2 = \frac{1}{(N+1)^2} \sum_{i=0}^{N-1} \sum_{N}^{j=i+1} \langle (\vec{r}_i - \vec{r}_j)^2 \rangle \tag{4.6}$$

olarak tanımlanır.

4.2 Monte Carlo simulasyon yöntemi

Monte Carlo simulasyon yöntemi bir problemi çözmek için rasgele sayıları kullanan nümerik bir tekniktir. Tarihte ilk büyük ölçekli Monte Carlo çalışması yirminci yüzyılın ortalarında gerçekleşmiştir. Stanislav Ulam, John von Neumann ve Enrico Fermi, Monte Carlo yöntemini pratik problemlerin çözümü için nümerik bir teknik olarak ilk kez önermiş ve uygulamışlardır. Monte Carlo yöntemi üzerine ilk yayın 1949 yılında Metropolis ve Ulam tarafından yayınlanan bilimsel makaledir[42].

Bu tarzda Monte Carlo simulasyonu gerçekleştirmek için rasgele, birbirinden bağımsız, reel ve düzgün olarak dağılmış (sıfırdan bire kadar) bir dizi sayı gereklidir. İlk Monte Carlo çalışmalarında rasgele sayı tabloları kullanılmıştır. Bu rasgele sayılar gerçek sargele işlemlerden (radyoaktif bozunma veya elektronik aygıtlardaki termal gürültü gibi) doğrudan üretilen sayılardır. Bununla birlikte böyle bir yaklaşım ancak Monte Carlo hesaplamaları elle gerçekleştirildiği müddetçe uygulanabilir[43].

Bilgisayar hesaplamaları için herhangi önceden hazırlanmış tabloların kullanımı pratik değildir. Bu sebeple ihtiyaç duyulduğu zaman basit aritmetik işlemleri uygulayan belli hesaplamalar ile rasgele sayılar üretilir. Deterministik algoritmalar ile üretilen bu sayılar bu sebeble tahmin edilebilir ve tekrar üretilebilir niteliktedirler. Eğer bu sayılar oldukça uzun bir tekrarda düzgün dağılmışlar ve bağımsız iseler bu sayılar **sözde-rasgele (pseudo-random)** olarak değerlendirilebilirler. Bununla birlikte, belli parametre setlerini kullanan belli rasgele sayı üreticileri tarafından üretilen rasgele sayıların kalitesinin çok farklı olabileceği akılda tutulmalıdır. Yani gerçek simulasyonlarda kullanmadan önce rasgeleliğin standart testi ile bu üreticilerin kalitesi kontrol edilmelidir[44].

4.2.1 Temel tanım

İstatistik fizik serbestlik derecesi çok olan sistemlerle uğraşır. İstatistik fizikteki tipik görevlerden biri sistemin makroskobik gözlenebilir niceliklerinin ortalama değerlerini hesaplamaktır. Kanonik bir toplulukta, A(**x**) gözlenebilir niceliğinin termal ortalaması

$$\langle A(\mathbf{x})\rangle_T = \frac{1}{Z}\int d\mathbf{x}\exp\left[-\mathcal{H}(\mathbf{x})/k_BT\right]A(\mathbf{x}), \tag{4.7}$$

şeklinde tanımlanır; burada Z

$$Z = \int d\mathbf{x}\exp\left[-\mathcal{H}(\mathbf{x})/k_BT\right]. \tag{4.8}$$

ile verilen bölüşüm fonksiyonudur (burada k_B Boltzmann sabiti ve T sıcaklıktır.). Bu ifadelerde **x**, bir noktayı faz uzayında tam olarak belirlemek için gerekli koordinatlar takımını gösterir. N-parçacıklı bir sistem için

$$d\mathbf{x} = dx_1\,dy_1\,dz_1\,dx_2\ldots dz_N\,dp_{x_1}\,dp_{y_1}\,dp_{z_1}dp_{x_2}\ldots dp_{z_N}. \tag{4.9}$$

dir. Normalize Boltzmann faktörü

$$p(\mathbf{x}) = \frac{1}{Z}\exp\left[-\mathcal{H}(\mathbf{x})/k_BT\right] \tag{4.10}$$

termal dengede **x** konfigürasyonunun meydana geldiği istatistiksel ağırlığı anlatan olasılık yoğunluğu rolünü oynar.

Denge istatistik mekanikte Monte Carlo yöntemi Denklem 4.7'ye yaklaşma fikrinden başlar; burada istatistiksel örnek olarak kullanılan $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M\}$ faz uzayı noktalarının karakteristik bir alt kümesini kullanarak sonlu bir integrasyon ile $p(\mathbf{x})$ ağırlıklı tüm {**x**} durumları üzerinden integral alınır. Açıkça $M \longrightarrow \infty$ limitinde

$$\langle A(\mathbf{x})\rangle = \frac{\sum_{l=1}^{M}\exp\left[-\mathcal{H}(\mathbf{x_l})/k_BT\right]A(\mathbf{x_l})}{\sum_{l=1}^{M}\exp\left[-\mathcal{H}(\mathbf{x_l})/k_BT\right]} \tag{4.11}$$

toplamı Denklem 4.7'ye yaklaşmak zorundadır; çünkü burada integral işareti toplam işareti ile yer değiştirmiştir..

4.2.2 Basit örnekleme (Simple sampling)

$f(x)$'in çok boyutlu **x** vektörünün yerine sadece bir x reel değişkeninin bir fonksiyonu olduğu $\int f(x)dx$ tek boyutlu integrallerini çözen standart rutinlerden farklı olarak burada $\mathbf{x}_l$ noktalarını düzgün bir ızgara'ya (grid) göre seçmenin bir anlamı yoktur[45]. Izgara

noktalarının çok daha tekdüze (uniform) ve güzel dağılımı, şayet $\mathbf{x_l}$ noktaları rasgele seçilirse elde edilir. Bu Monte Carlo yöntemi **basit örnekleme** (simple sampling) olarak adlandırılır[44]. Bu, rasgele yürümeler, self-avoiding walk yürüme ve süzülme (percolation) gibi termal olmayan problemler için çok güzel çalışmaktadır[46]. Basitliği yanında basit örnekleme yönteminin en önemli avantajları şunlardır:

i. her bir konfigürasyon birbirinden istatistiksel olarak bağımsızdır ve bu yüzden standart hata analizleri kullanılır,

ii. zincirimsi problemler için bir simulasyon koşmasında (RUN) N zincir uzunluğu için belli bir uzunluğa kadar bilgi elde edilir.

4.2.3 Önem örnekleme (Importance sampling)

Denklem 4.11 gibi termal ortalamaları hesaplamak istenildiğinde basit rasgele örnekleme teknikleri kullanışlı değildir. Örneğin Denklem 4.11'de $A(\mathbf{x}_l)$'ın kendisi Hamiltonyen $\mathcal{H}(\mathbf{x}_l)$ olsun. Bu durumda, zincirin bağı başına C spesifik ısısı

$$\frac{C}{k_B} = \frac{1}{N}\frac{\partial\left(\langle\mathcal{H}\rangle_T\right)}{\partial\left(k_BT\right)} = \frac{\langle\mathcal{H}^2\rangle_T - \langle\mathcal{H}\rangle^2}{N(k_BT)^2}, \tag{4.12}$$

ile verilir; bu denklem

$$\frac{\langle\mathcal{H}^2\rangle_T - \langle\mathcal{H}\rangle^2}{\langle\mathcal{H}\rangle_T^2} \propto \frac{1}{N}. \tag{4.13}$$

bağıl dalgalanmasını ima eder. Bu sebeble büyük N için serbestlik derecesi başına E enerjisinin $p(E)$ olasılık dağılımı

$$p(E) = \frac{1}{N}\int d\,\mathbf{x}\,\delta\left(\mathcal{H}(\mathbf{x}) - NE\right)\exp\left[-\mathcal{H}(\mathbf{x})/k_BT\right] \tag{4.14}$$

şekline gelir ve bu dağılım çok keskin şekilde pik yapar.

$$\langle\mathcal{H}\rangle_T = N\int_{-\infty}^{+\infty} E\,p(E)\,dE, \qquad \langle\mathcal{H}^2\rangle_T = N\int_{-\infty}^{+\infty} E^2\,p(E)\,dE \tag{4.15}$$

ile $p(E)$'nin $E = \langle\mathcal{H}\rangle_T/N$ civarında pik yüksekliğinin $\sqrt{N}$ ve pik genişliğinin $1/\sqrt{N}$ olacağı gözükür. İkinci ve üçüncü dereceden faz geçişlerinde $p(E)$ Gaussian'dır. Bu sebeble basit örnekleme kullanarak $\langle\mathcal{H}/N\rangle$ yakınındaki bölgede E'li durumları üretme olasılığı exponansiyel olarak küçüktür. Yani diğer bir deyişle $\{x_l\}$ faz uzayı noktaları Denklem 4.11'e exponansiyel olarak küçük $p(x)$ ağırlık faktörleri ile girerler. Bu durumda ihtiyaç duyulan şey Denklem 4.11'de içerilen x_l konfigürasyonlarını tamamen rasgele değil, fakat T sıcaklığında tercih edilebilir şekilde önemli olarak örnekleyen daha etkili bir

tekniktir[47]. $\mathbf{x_l}$ faz uzayı noktalarının bir $P(\mathbf{x_l})$ olasılığına göre seçilmiş bir işlemi göz önüne alalım. Bu durumda ortalama ifade aşağıdaki şekilde düzeltilmelidir.

$$\langle A(\mathbf{x})\rangle = \frac{\sum_{l=1}^{M} \exp\left[-\mathcal{H}(\mathbf{x}_l)/k_BT\right] A(\mathbf{x}_l)/P(\mathbf{x}_l)}{\sum_{l=1}^{M} \exp\left[-\mathcal{H}(\mathbf{x}_l)/k_BT\right]/P(\mathbf{x}_l)}. \tag{4.16}$$

$P(\mathbf{x}_l)$'e ait basit ve çok doğal bir seçim, $P(\mathbf{x}_l) \propto \exp\left[-\mathcal{H}(\mathbf{x}_l)/k_BT\right]$ şeklinde olacaktır. O zaman Boltzmann faktörleri birbirini götürür, Denklem 4.16 basit bir aritmetik ortalamaya indirger.

$$\langle A(\mathbf{x})\rangle = \frac{1}{M}\sum_{l=1}^{M} A(\mathbf{x}_l). \tag{4.17}$$

Bu tür örneklemeye **önem örnekleme** (importance sampling) denilir[43]. Metropolis ve çalışma arkadaşları $\{x_l\}$ birbirini takip eden durumlarını birbirinden bağımsız olarak seçmemeyi fakat

$$W(\mathbf{x_l} \rightarrow \mathbf{x}_{l+1}). \tag{4.18}$$

uygun geçiş olasılığı ile $\{\mathbf{x}_{l+1}\}$ durumunun kendinden önceki $\{\mathbf{x}_l\}$ durumundan oluşturulmasını önerdiler. Onlar ayrıca $M \rightarrow \infty$ limitinde Markov süreci ile üretilen durumların dağılım fonksiyonunun

$$P_{\text{eq}}(\mathbf{x}_l) = \frac{1}{Z}\exp\left(-\frac{\mathcal{H}(\mathbf{x}_l)}{k_BT}\right) \tag{4.19}$$

denge dağılımına doğru gittiği W geçiş olasılığını seçmenin mümkün olduğunu belirttiler. Bunu gerçekleştirmenin yeterli bir koşulu detaylı denge prensibi

$$P_{\text{eq}}(\mathbf{x}_l)W(\mathbf{x}_l \rightarrow \mathbf{x}_{l'}) = P_{\text{eq}}(\mathbf{x}_{l'})W(\mathbf{x}_{l'} \rightarrow \mathbf{x}_l). \tag{4.20}$$

eşitliğini kabul etmektir. Denklem 4.20 açıkça $\mathbf{x}_l \rightarrow \mathbf{x}_{l'}$'yi eşsiz şekilde belirtmez W'nin açık seçiminde bazı keyfilikler kalır. En sık kullanılan seçim [48, 49]

$$W(\mathbf{x}_l \rightarrow \mathbf{x}_{l'}) = \begin{cases} \exp\left(-\delta\mathcal{H}/k_BT\right) & \text{eğer} \quad \delta\mathcal{H} > 0 \quad \text{ise,} \\ \\ 1 & \text{aksi hallerde.} \end{cases} \tag{4.21}$$

olarak verilir.

Bu tarzda önem örnekleme algoritması aşağıdaki gibidir: Herhangi bir zamanda sistemin o anki durumu *durum-0* ile gösterilir. Sistemin ϵ_0 enerjisine ve A_0 özellik değerine sahiptir. Rasgele bir hareketten sonra sistem yeni bir deneme durumunda (*durum-1*) ϵ_1 enerjisi ve A_1 özellik değeri ile olacaktır. Simulasyon şu şekilde devam eder. Bu deneme durumu kabul edilsin veya red edilsin bu girişim bir Monte Carlo adımı olarak hesaba katılır. Kabul kriteri Denklem 4.21'de verildiği gibidir; yani

- Eğer $\epsilon_1 < \epsilon_0$ ise: deneme durumu 1 olasılığı ile kabul edilir ve yeni *durum-0* olur. Ayrıca yeni durumun özelliği ($A_{\text{Toplam}} = A_{\text{Toplam}} + A_1$) toplamına eklenir.

- Eğer $\epsilon_1 > \epsilon_0$ ise: deneme durumu $\exp[-(\epsilon_1 - \epsilon_0)/k_BT]$ olasılığı ile kabul edilir yani kabul edilirse deneme durumu yeni *durum-0* olur. Eğer red edilirse o zaman sistem eski *durum-0*'ında kalır. Teknik olarak, $\exp[-(\epsilon_1 - \epsilon_0)/k_BT]$ olasılığı ile kabul onu $0 \leq \zeta \leq 1$ rasgele bir sayı ile eşleştirilerek çözülür. Eğer $\exp[-(\epsilon_1 - \epsilon_0)/k_BT] > \zeta$ ise deneme durumu kabul edilir

Not etmek gerekir ki önem örnekleme yapıldığında farklı durumlar birbirlerinden bağımsız değillerdir. Bu sebeple standart hata analizleri kullanılamaz.

4.3 Problem ve çözüm yöntemi

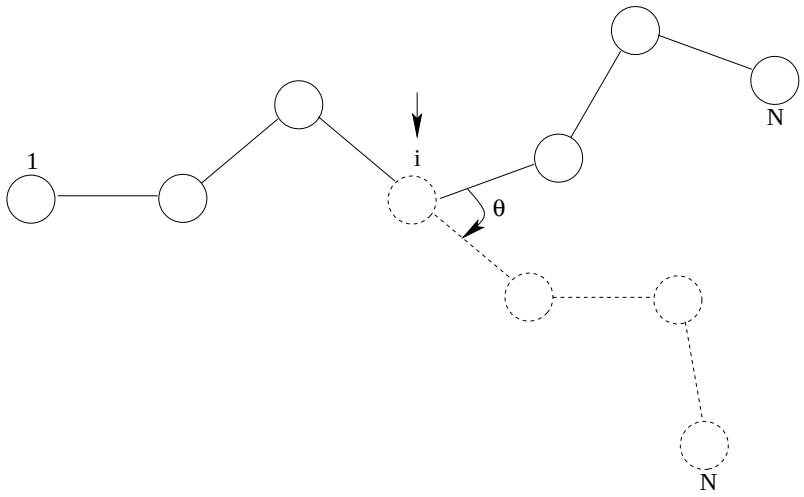
Paralel hesap yardımı ile incelenmek istenen sistem, N tane parçacıktan oluşmuş olan ideal bir polimer zinciridir. Bu polimer zincirinin davranışı teorik olarak öngörülmüştür (bakınız Denklem 4.5). Ayrıca deneysel ve simulasyon çalışmalarında bu teorik öngörü ile aynı sonuçlar elde edilmiştir.

İdeal polimer zincirinin Monte Carlo simulasyonunda sistem keyfi bir konfigürasyondan başlatılır ve uygun Monte Carlo adımları ile sistemin dengeye gelmesi sağlanır. Bu tür simulasyonlarda kullanılan belli başlı Monte Carlo hareketleri literatürde mevcuttur[50, 51] (örneğin pivot hareketi, yerel monomer hareketi, polimerin yılan gibi hareket etmesi). Bu çalışmamızda sistemi dengeye getirmek için sadece pivot hareketinin yeterli olacağını düşünerek Monte Carlo adımı olarak pivot hareketi uyguladık.

Pivot hareketi: Pivot hareketi rasgele yürüme hareketi ile özdeş bir harekettir. Pivot hareketinde N tane monomerden bir tanesi rasgele olarak seçilir ve polimer zincirinin rasgele seçilen monomere göre sağ kolu (seçilen monomerden sonuncu monomere kadar olan zincir parçası) küresel koordinatların θ ve ϕ açıları ile rasgele miktarlarda döndürülür. Elde edilen yeni konfigürasyon Metropolis kriteri (bakınız Denklem 4.21) ile değerlendirilir[1]. Pivot hareketinin temsili Şekil 4.2'de gösterilmektedir[52].

Yapılan çok sayıdaki Monte Carlo adımlarından sonra sistemin dengeye geldiği, sisteme ait bazı niceliklerin (örneğin uçtan uca mesafe, jirasyon yarıçapı) gözlenmesi ile tespit edilir. Bu tespitten sonra sisteme ait niceliklerin ortalama değerlerinin hesaplanmasına geçilir. Simulasyon sonucunda elde edilen ortalama değerler teorik öngörüler ile karşılaştırılır.

[1] İdeal polimer zincirinde etkileşimler olmadığından her bir yeni durum bir önceki ile aynı olasılıktadır ve sistem yeni duruma geçer.



Şekil 4.2: Pivot hareketi.

Biz bu çalışmamız boyunca yazdığımız program kaynak kodlarında pivot hareketi gerçekleştiren fonksiyonu **“pivot()”**, metropolis kriterini kontrol eden fonksiyonu **“metropolis()”**, uçtan uca mesafeyi hesaplatan fonksiyonu **“Rendtoend()”** ve jirasyon yarıçapını hesaplatan fonksiyonu **“Gyration()”** ve yeni bir konfigürasyonu dosyadan okuyan fonksiyonu **“read_coords()”** ve girdi parametrelerini (simulasyon için gerekli bazı değerler) dosyadan okuyan fonksiyonu **“read_inputs()”** olarak gösterdik. Bu fonksiyonların C dilindeki bildirimleri aşağıdaki gibidir:

int pivot(void)

int metropolis(void)

int Rendtoend(int deger)

int Gyration(int deger)

read_inputs(char *dosyaismi)

read_coord(char *dosyaismi)

4.4 Seri hesap ve sonuçlar

Bu bölümde ideal polimer zincirinin Monte Carlo simulasyonunun seri hesabına ait detaylar ve sonuçlar verilmektedir.

4.4.1 Hesap detayları

Simulasyon programının kaynak kodunda bulunan değişkenlerin bazıları şunlardır:

nummon : Zincirdeki parçacık veya monomer sayısı,

mcm : Monomer başına Monte Carlo adım sayısı,

relaxtime : Sistemi dengeye getirmek için harcanan Monte Carlo adım sayısı,

chlencyc : Zincir uzunluğuna bağımlılığı hesap ederken kullanılan değişken.

Seri hesaba ait algoritmik akış aşağıdaki gibidir:

1. Başla,
2. İlgili dizi ve değişkenleri tanımla,
3. Gerekli dizi elemanları ve/veya değişkenlerin değerlerini sıfırla,
4. İlgili veri okuma/yazma yapılacak olan dosyaları aç,
5. `read_inputs()` fonksiyonuyla ile dosyadan nummon, mcm, relaxtime değişkenleri ve dökümlerde kullanılmak üzere listeleme adım sayısı değerlerini oku,
6. `read_coord()` fonksiyonuyla dosyadan sisteme ait keyfi başlangıç konfigürasyonunu elde et,
7. Rasgele sayı yaratıcısını sıfırla,
8. Döngü içerisinde relaxtime değeri kadar pivot(), metropolis() ve Rendtoend() fonksiyonlarını çağırarak sistemin dengeye gelmesini bekle,
9. **mcm*nummon** değerine kadar dönecek bir döngü oluştur, bu döngü içerisinde pivot(), metropolis() fonksiyonlarını çağır, zincir uzunluğuna bağımlılığı hesap etmek üzere mcm defa Rendtoend() ve Gyration() fonksiyonları ile uçtan uca mesafe ve jirasyon yarıçaplarını hesapla, ortalama değerler için gerekli toplamları yap,
10. Ortalama değerleri hesapla,
11. Ortalama değerleri ilgili dosyalara yaz,
12. Dosyaları kapat,
13. Dur.

Bu algoritmaya ait kaba kod (“pseudo kod”) aşağıda verilmektedir.

```
1 BEGIN
2    DEFINE ilgili tum degisken ve diziler
3    SET ilgili tum degiskenler ve diziler TO Sifir
4    SET FILE ilgili dosyalar
5
6    SET  timevalue TO  Sifir
7
8    CALL DosyaAc();     /// ilgili dosyalari okuma-yazma yapmak icin ac.
9
10   CALL read_inputs(mcm,nummon,chlencyc,relaxtime);
                        // Girdi parametrelerini oku.
11
12   CALL read_coord(x,y,z);   // Baslangic koordinatlarini oku.
13
14   CALL srand(time(0));  // Rasgele sayi yaraticisini sifirla.
15
16   SRunAve=0.0;
17
18   FOR (i = 1; i<=relaxtime; i++)
19
20      CALL pivot();
21      CALL metropolis();
22      CALL Rendtoend(nummon);
23      SRunAve = SRunAve + ree2;
24
25      WRITE ilgili dosyaya (i,sqrt(ree2),sqrt(SRunAve/(double)i));
26
27   ENDFOR
28
29   FOR (k = 1; k<=30; k++)
30      SRee2[k]=0.0;
31      SRgy2[k]=0.0;
32   ENDFOR
33   CALL gettimeofday(t_ilk); // Zaman tutmaya basla.
34
35   //// ANA DONGU ////
36
37   FOR (i = 0; i<=mcm; i=i+1)
38
39        FOR (j = 0; j<nummon; j=j+1)
40
41           CALL pivot();
```

```
42           CALL metropolis();
43
44        ENDFOR          // For (j<nummon) bitti
45
46     FOR (il = chlencyc; il<=nummon; il+=chlencyc)
47
48        CALL Rend2end(il);
49        CALL Gyration(il);
50
51        SRee2[il/chlencyc]+=ree2;
52        SRgy2[il/chlencyc]+=rgy2;
53     ENDFOR
54
55   ENDFOR    // For (i<=mcm) bitti
56
57   CALL  gettimeofday(t_son);  // Zaman tutmayi bitir.
58   saniye = t_dif.tv_sec = t_son.tv_sec - t_ilk.tv_sec;
59
60   gecensure=saniye;
61
62   kmax=nummon/chlencyc;
63
64   FOR (k = 1; k<=kmax; k++)
65      WRITE ilgili dosyaya (k*chlencyc,sqrt(sree2[k]/(double)mcm));
66      WRITE ilgili dosyaya (k*chlencyc,sqrt(srgy2[k]/(double)mcm));
67   ENDFOR
68
69   WRITE ilgili dosyaya (isim1,nummon,isim2,mcm,isim3,
                           chlencyc,isim4,relaxtime);
70   WRITE ilgili dosyaya (gecensure);
71
72   CALL DosyaKapat();   // Acilan tum dosyalari kapatir.
73 END.
```

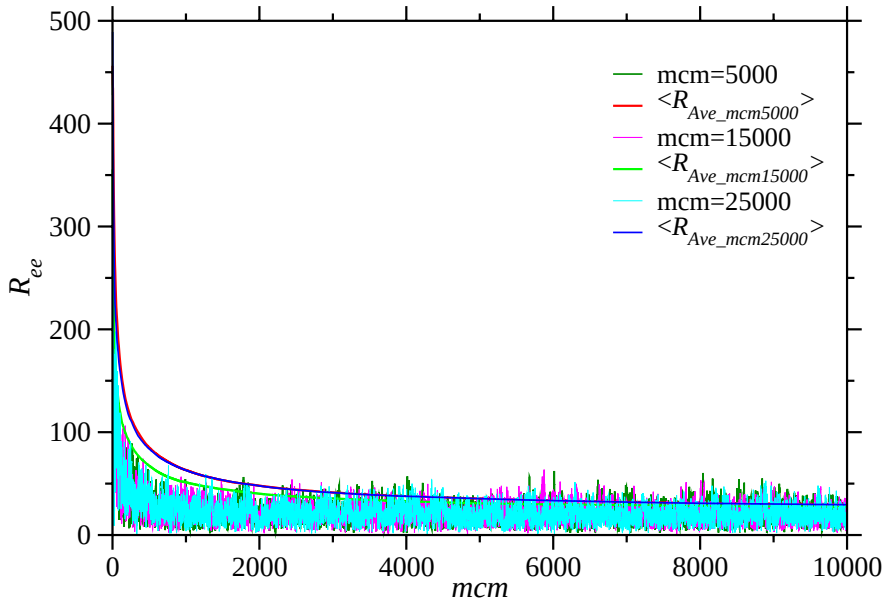
4.4.2 Hesap sonuçları

Bu çalışmada iki farklı büyüklükte ideal polimer zincirinin davranışı simule edilmiştir. Zincir boyları 480 ve 960 olarak ayrı ayrı incelenmiştir. Buradaki gaye sistem büyüdüğü zaman paralel hesabın önemini görmeye yöneliktir. Ayrıca teorik öngörüler mümkün olduğunca büyük sistemler için geçerlidir.

N=480 ve N=960 için ideal polimer zincirlerinin simulasyonlarında mcm=5000, mcm=15000 ve mcm=25000 için simulasyonlar yapılmıştır. Ortalama değerleri hesaplamaya başlamadan önceki zaman diliminde uçtan uca mesafeler anlık olarak gözlemlenmiş

ve sistemin dengeye geldiği teyid edilmiştir. Uçtan uca mesafe ve jirasyon yarıçapı farklı zincir uzunluklarına bağımlılık için ölçülmüştür. Burada not edilmesi gereken nokta ise örneğin N=960 olduğu durumda aynı zincir boyu üzerinde 40, 80, 120, ... şeklinde alt küçük zincirler göz önüne alınmıştır.

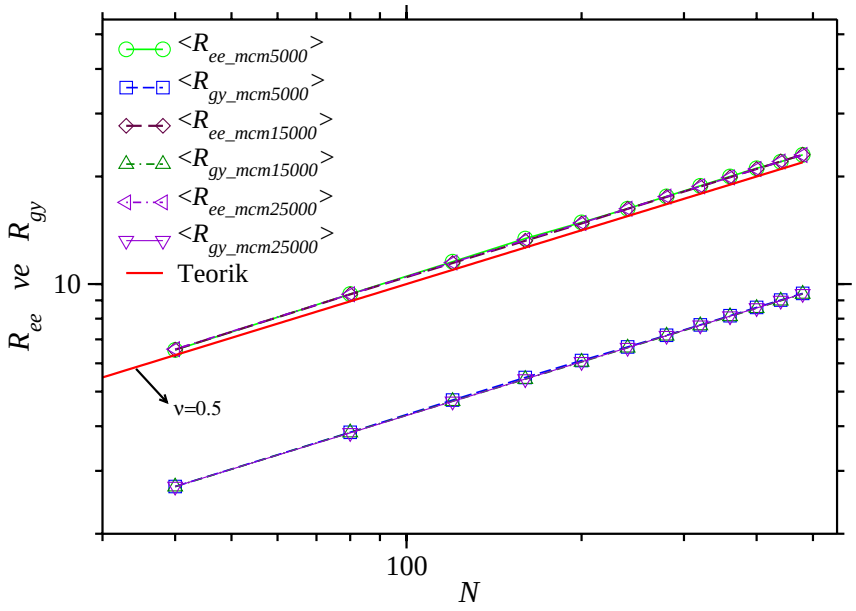
A) Zincir uzunluğu 480 monomer olduğu durum:
Şekil 4.3'de mcm=5000, mcm=15000 ve mcm=25000 olan simulasyon hesaplarında ortalama değerler alınmaya başlamadan önceki zaman diliminde sistemin dengeye geldiğini doğrulayan uçtan uca mesafe grafiği verilmektedir.



Şekil 4.3: Seri hesapta N=480 iken üç farklı mcm değerinde sistemin bir denge du rumuna eriştiğinin gözlenmesi.

Şekil 4.3'den görüldüğü üzere sistem başlangıçta nispeten açılmış vaziyettedir. İlerleyen Monte Carlo adımlarında sistem kıvrılmış kangal yapı (**"coil"**) almaktadır. Şekilde ayrıca uçtan uca mesafe ve jirasyon yarıçapı için yinelenen ortalama ("running average") da gösterilmektedir.

Şekil 4.4 ise uçtan uca mesafe ve jirasyon yarıçapının zincir boyuna bağımlılığı gösterilmektedir. Grafikte ayrıca teorik asimptotik davranış ($\nu = 0.5$) belirtilmektedir. Grafiklerden anlaşıldığı üzere hem uçtan uca mesafe hem de jirasyon yarıçapı zincir boyunun karekökü ile asimptotik davranış göstermektedir. Bu hesaba ait geçen süreler Tablo 4.1'de verilmektedir.



Şekil 4.4: Seri hesapta N=480 iken $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.

B) Zincir uzunluğu 960 monomer olduğu durum:

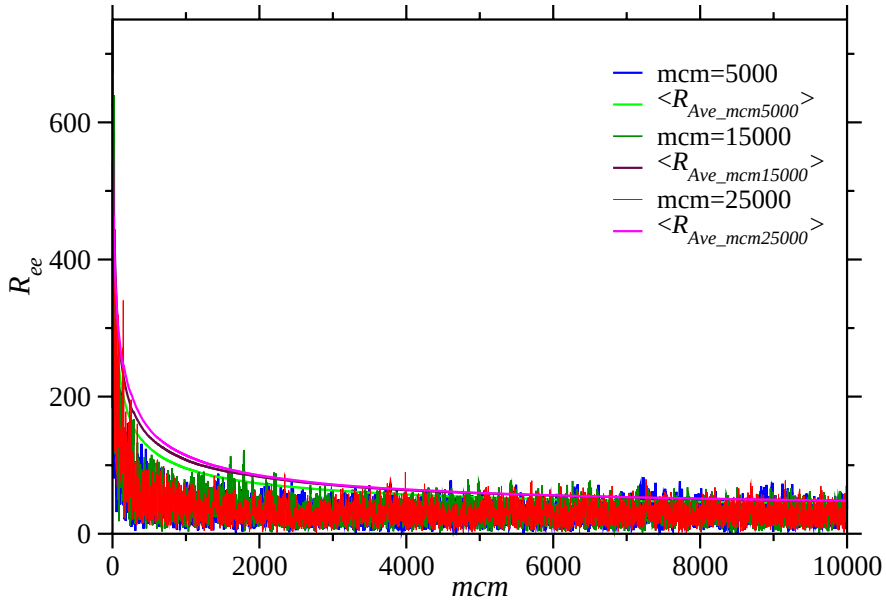
Zincir uzunluğunun 480 olduğu duruma benzer şekilde 960 olan durum içinde önce sistemin dengeye gelmesi beklenilmiştir (bakınız Şekil 4.5)

Şekil 4.6'e ise zincir boyunun 960 olduğu duruma ait uçtan uca mesafe ve jirasyon yarıçapının zincir boyu ile değişimi verilmektedir. Sonuçlar yine teorik öngörüler ile tam bir uyum içerisindedir.

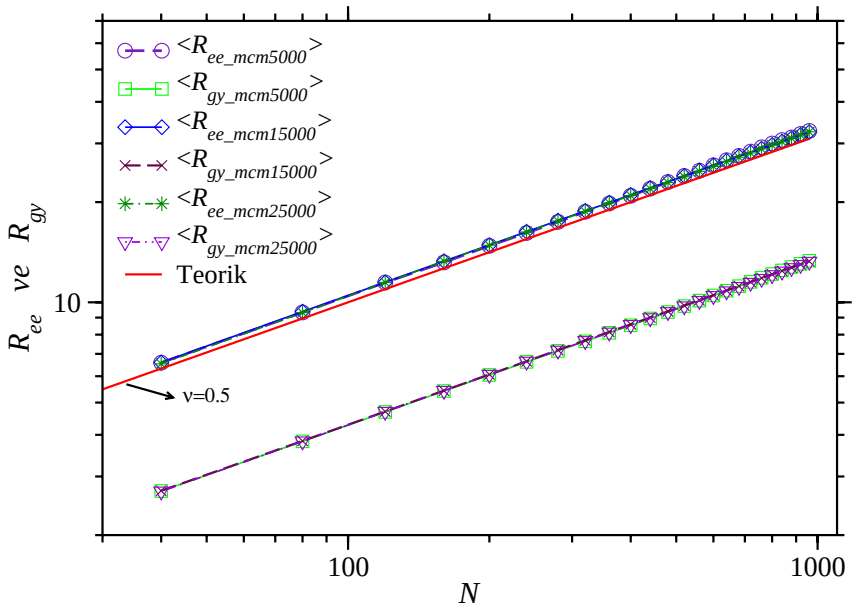
Bu hesaba ait geçen süreler ise yine Tablo 4.1'de verilmektedir. Tablodan görüldüğü üzere sistem büyüdükçe geçen süre lineer olmayan tarzda artmaktadır. Diğer yandan aynı sistem büyüklüğünde geçen süre Monte Carlo adım sayısı ile lineer olarak artmaktadır.

	N=480 için geçen süre (saniye)			**N=960 için geçen süre (saniye)**		
n_p	**mcm=5000**	**mcm=15000**	**mcm=25000**	**mcm=5000**	**mcm=15000**	**mcm=25000**
1	867	2602	4336	3740	11202	18642

Tablo 4.1: Seri hesap için zincir uzunluğu 480 ve 960 monomer olduğu durumda geçen süreler.



Şekil 4.5: Seri hesapta N=960 iken üç farklı mcm değerinde sistemin bir denge du rumuna eriştiğinin gözlenmesi.



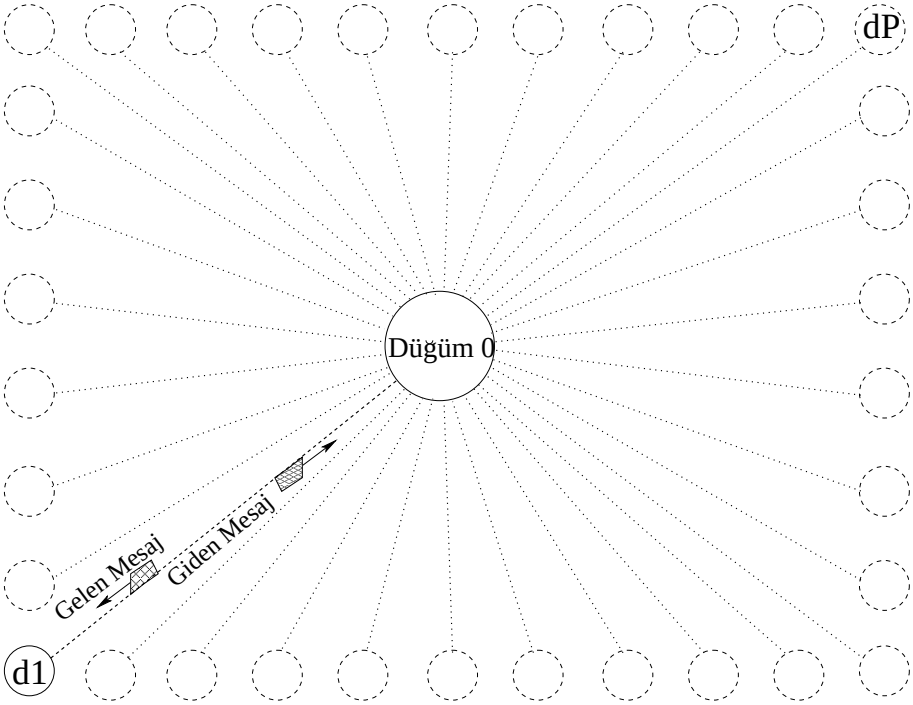
Şekil 4.6: Seri hesapta N=960 iken $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.

4.5 Paralel hesap ve sonuçlar

Bu bölümde önceki bölümde seri hesabı yapılan sistemin paralelleştirilmesi gerçekleştirilmekte ve sonuçlar gösterilmektedir.

4.5.1 Hesap detayları

Problemin paralelleştirilmesinde **"Yönetici-işçiler"** yaklaşımı uygulanmıştır. Yapılan hesapta işçilere yük dağılımı statik olarak gerçekleştirilmiştir. Yapılan algoritmada yönetici makine sistemin dengeye gelmesini sağlamakta ve daha sonra ana simulasyon kısmını başlatmaktadır. Döngüler içerisindeki pivot ve metropolis işlemleri de yapıldıktan sonra elde edilen yeni konfigürasyon işçilere bildirilir. İşçi makineler ise kendilerine bildirilen yeni konfigürasyonlar için kendilerine ait farklı zincir uzunlukları için uçtan uca mesafeleri ve jirasyon yarıçapları hesaplayarak sonuçları yönetici makineye bildirmektedirler. Hesaptaki iletişim Şekil 4.7'de temsil edilmektedir.



Şekil 4.7: Düğüm 0 (kök süreç) ile diğer düğümler arasındaki iletişim.

Şekil 4.7'de görüldüğü gibi, çok işlemcili programda düğüm 0 haricindeki düğümler arasında iletişime izin verilmemiştir. Bu hem senkronizasyon süresini, hem de tampon-

lanmayı en aza indirgemek için uygulanmıştır. Buradaki şekilde P adet süreç makinesi (düğüm) bulunmaktadır. Bunlar d_1 den başlayarak d_P'ye kadar sadece düğüm 0 makinesi, yani ana süreçteki makineyle iletişimdedirler.

Paralel hesapta toplam en fazla 25 işlemci hesaba dahil olmuştur. Makinelerin tümü özdeş olup teknik özellikleri "Intel Pentium 4 (3.2 GHz) işlemcili, 512 MB RAM kapasiteli, Linux işletim sistemi tabanlı" olan bilgisayarlardır. Tüm bilgisayarlar 10/100 Mbit/s hızla Anahtar cihazı üzerinden birbirine bağlanmıştır. Paralel hesaplama daha önce söylenildiği üzere C dilinde MPI kütüphanesini kullanılarak Linux ortamında gerçekleştirilmiştir [2].

Tüm bunların yapmış oldukları işler aynı fakat kullandıkları veriler farklı olduğu için paralel hesapta **alan ayrışımı** ("domain decomposition") yapılmıştır. İşlemin tümü aynı ağırlıkta ve makinalar aynı güçte olduklarından özel bir haritalamaya gerek duyulmamıştır.

Yukarıda anlatılan bu hazırlıklardan sonra seri kod, MPI kütüphanesini kullanabilir hale getirilmiştir, bu sayede paralel hesaplama ve iletişim sağlanmıştır. Bu iş için gerekli olan **"mpi.h"** başlık dosyası eklenmiş ve main() isimli programın çalışan ana fonksiyonu, mesaj-geçme iletişimini yapacak çeşitli fonksiyonların ilgili yerlerde kullanımıyla şekillendirilerek yapısal değişikliklere gidilmiştir.

Paralel hesaba ait algoritmik akış aşağıdaki gibidir:

1. Başla,
2. İlgili dizi ve değişkenleri tanımla,
3. Gerekli dizi elemanları ve/veya değişkenlerin değerlerini sıfırla,
4. MPI'ı başlat,
5. İşlemci sayısı ve kimliklerini tespit et,
6. Rasgele sayı yaratıcısını sıfırla,
7. Eğer **Yönetici** makine ise;
    i. Parametreleri oku,
    ii. İlgili dosyaları veri okuma/yazma için aç,
    iii. Başlangıç konfigürasyonunu al,
    iv. Sistemi dengeye getir,

[2] MPI için LAM/MPI uyarlaması kullanımı, program derlenmesi ve çalıştırılması için EK B'ye bakınız.

v. Simulasyonu başlat, her bir yeni konfigürasyonu işçilere bildir,

vi. İşçilerden sonuçları topla,

8. Eğer **İşçi** makine ise;

i. Yöneticiden konfigürasyonları al,

ii. Kendine ait farklı zincir boylarında uçtan uca mesafeleri ve jirasyon yarıçaplarını hesap et,

iii. Sonuçları yöneticiye gönder,

9. Eğer **Yönetici** isen dosyaları kapat,

10. MPI bitir,

11. Dur.

Bu algoritmaya ait kaba kod (“pseudo kod”) aşağıda verilmektedir.

```
1 BEGIN
2   DEFINE ilgili tum degisken ve diziler
3   SET ilgili tum degiskenler ve diziler TO Sifir
4   SET FILE ilgili dosyalar
5
6   SET  timevalue TO  Sifir
7   DEFINE TAGA 100, TAGB 200, TAGC 300, TAGD 400,
           TAGE 500, TAGF 600, TAGG 700, TAGH 800
8
9   MPI_Init(&argc, &argv);
10  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12  MPI_Get_processor_name(processor_name,&namelen);
13
14  CALL  srand(time(0));  // Rasgele sayi yaraticisini sifirla.
15
16  IF (rank=0) THEN
17
18   CALL DosyaAc();     /// ilgili dosyalari okuma-yazma yapmak icin ac.
19   CALL read_inputs(mcm,nummon,adim);  // Girdi parametrelerini oku.
20   CALL read_coord(x,y,z);   // Baslangic koordinatlarini oku.
21
22   SRunAve=0.0;
23
```

```
24  FOR (i = 1; i<=relaxtime; i++)
25
26      CALL pivot();
27      CALL metropolis();
28      CALL Rendtoend(nummon);
29      SRunAve+=ree2;
30      WRITE ilgili dosyaya (i,sqrt(ree2),sqrt(SRunAve/(double)i))
31
32  ENDFOR
33
34  startwtime = MPI_Wtime(); //Surec tabanli olarak zaman tutmaya basla
35
36  //// ANA DONGU ////
37
38  FOR (i = 0; i<=mcm; i=i+1)
39
40   FOR (say=1; say<=numprocs-1;  say = say + 1)
41      MPI_Ssend(&nummon,1,MPI_INT,say,1+say,MPI_COMM_WORLD);
42      MPI_Ssend(&mcm,1,MPI_INT,say,2+say,MPI_COMM_WORLD);
43      MPI_Ssend(&chlencyc,1,MPI_INT,say,3+say,MPI_COMM_WORLD);
44      MPI_Ssend(&i,1,MPI_INT,say,TAGA+say,MPI_COMM_WORLD);
45   ENDFOR
46
47   FOR (j = 0; j<nummon; j=j+1)
48
49      CALL pivot();
50      CALL metropolis();
51
52   ENDFOR          // For (j<nummon) bitti
53
54   FOR (say = 1; say<=numprocs-1; say = say + 1)
55      MPI_Ssend(x,nummon,MPI_DOUBLE,say,TAGB+say,MPI_COMM_WORLD);
56      MPI_Ssend(y,nummon,MPI_DOUBLE,say,TAGC+say,MPI_COMM_WORLD);
57      MPI_Ssend(z,nummon,MPI_DOUBLE,say,TAGD+say,MPI_COMM_WORLD);
58   ENDFOR
59
60   FOR (say = 1; say<=numprocs-1; say++)
61      kmax=nummon/chlencyc;
62      FOR(k=say;k<=kmax;k+=(numprocs-1))
63      MPI_Recv(&sree2[k],1,MPI_DOUBLE,say,TAGE+say,
                              MPI_COMM_WORLD,&status);
64      MPI_Recv(&srgy2[k],1,MPI_DOUBLE,say,TAGF+say,
                              MPI_COMM_WORLD,&status);
65      ENDFOR
66   ENDFOR
```

```
67
68   ENDFOR    // For (i<=mcm) bitti
69
70   ELSE
71
72   FOR ( ; ; )
73
74      MPI_Recv(&nummon,1,MPI_INT,0,1+rank,MPI_COMM_WORLD,&status);
75      MPI_Recv(&mcm,1,MPI_INT,0,2+rank,MPI_COMM_WORLD,&status);
76      MPI_Recv(&chlencyc,1,MPI_INT,0,3+rank,MPI_COMM_WORLD,&status);
77      MPI_Recv(&i,1,MPI_INT,0,TAGA+rank,MPI_COMM_WORLD,&status);
78
79      MPI_Recv(x,nummon,MPI_DOUBLE,0,TAGB+rank,MPI_COMM_WORLD,
                  &status);
80      MPI_Recv(y,nummon,MPI_DOUBLE,0,TAGC+rank,MPI_COMM_WORLD,
                  &status);
81      MPI_Recv(z,nummon,MPI_DOUBLE,0,TAGD+rank,MPI_COMM_WORLD,
                  &status);
82
83      kmax=nummon/chlencyc;
84
85      FOR (k=rank; k<=kmax;  k+=(numprocs-1))
86         il=k*chlencyc;
87         Rendtoend(il);
88         Gyration(il);
89         SRee2[k]+=ree2;
90         SRgy2[k]+=rgy2;
91      ENDFOR
92
93   if(i==mcm) break;  // Sonsuz donguyu kirarak cikiyoruz.
94
95   ENDFOR    // Sonsuz dongu yani for( ; ; ) sonu...
96
97      FOR (k = rank; k<=kmax; k += (numprocs-1))
98         MPI_Send(&sree2[k],1,MPI_DOUBLE,0,TAGE+rank,MPI_COMM_WORLD);
99         MPI_Send(&srgy2[k],1,MPI_DOUBLE,0,TAGF+rank,MPI_COMM_WORLD);
100     ENDFOR
101
102  ENDIF    // if (rank=0) icin else kismi bitti
103
104  //Saat tutmayi bitir
105  endwtime = MPI_Wtime();
106  gecensure=endwtime-startwtime;
107
108  IF (rank=0) THEN
```

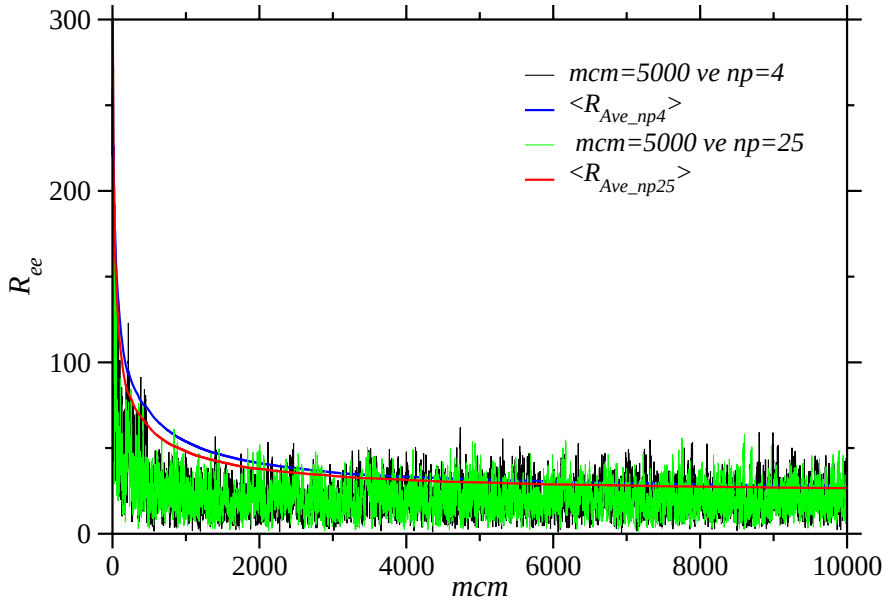
```
109
110    FOR (k = 1; k<=kmax; k++)
111       WRITE ilgili dosyaya (k*chlencyc,sqrt(sree2[k]/(double)mcm));
112       WRITE ilgili dosyaya (k*chlencyc,sqrt(srgy2[k]/(double)mcm));
113    ENDFOR
114
115  WRITE ilgili dosyaya (isim1,nummon,isim2,mcm,isim3,
                              chlencyc,isim4,relaxtime);
116  WRITE ilgili dosyaya (gecensure);
117
118  ENDIF
119
120  CALL DosyaKapat();   // Acilan tum dosyalari kapatir.
121
122  MPI_Finalize();
123
124 END.
```

4.5.2 Hesap sonuçları

A) Zincir uzunluğu 480 monomer olduğu durum:
Şekil 4.8 ve 4.9 ve 4.10'da zincirin uzunluğunun 480 olduğu sistemin dengeye ulaşırken yapılan gözlem verilmektedir. Burada aynı zincir uzunluğu farklı mcm değerleri için farklı işlemci sayıları (n_p) (2'den 25'e kadar) dahil ederek simule edilmiştir. Şekillerde görüldüğü üzere her bir farklı mcm ve n_p'de aynı sonuçlar elde edilmektedir. Sistem başlangıçta nispeten açılmış vaziyette iken daha sonra teorinin öngördüğü üzere kangal yapıdaki denge haline ulaşmaktadır (belli anlık görüntü için Şekil EK A.1'e bakınız).

mcm=5000, 15000, 25000 olan durumlarda n_p=4 ve n_p=25'e ait ortalama uçtan uca mesafelerin ve jirasyon yarıçapının zincir boyu değişimleri Şekil 4.11 ve Şekil 4.12 ve Şekil 4.13'de gösterilmiştir. Bu şekillerde ayrıca teorik öngörü ($\nu = 0.5$) de belirtilmektedir. Şekillerde elde edilen sonuçların her farklı durum için teorik öngörü ile uyum içinde olduğu gözlenmektedir.

B) Zincir uzunluğu 960 monomer olduğu durum:
Zincir uzunluğunun 480 olduğu duruma benzer şekilde paralel simulasyon gerçekleştirilmiştir (belli anlık görüntü için Şekil EK A.2'ye bakınız). Şekil 4.14'de mcm=25000 için n_p=4 ve n_p=25 olan durumda sistemin dengeye gelmesine ait gözlem verilmektedir. Şekil 4.15'de ise mcm=25000'de sistemin zincir boyuna bağımlılığı farklı işlemci sayıları altında gösterilmektedir. Burada elde edilen sonuç-



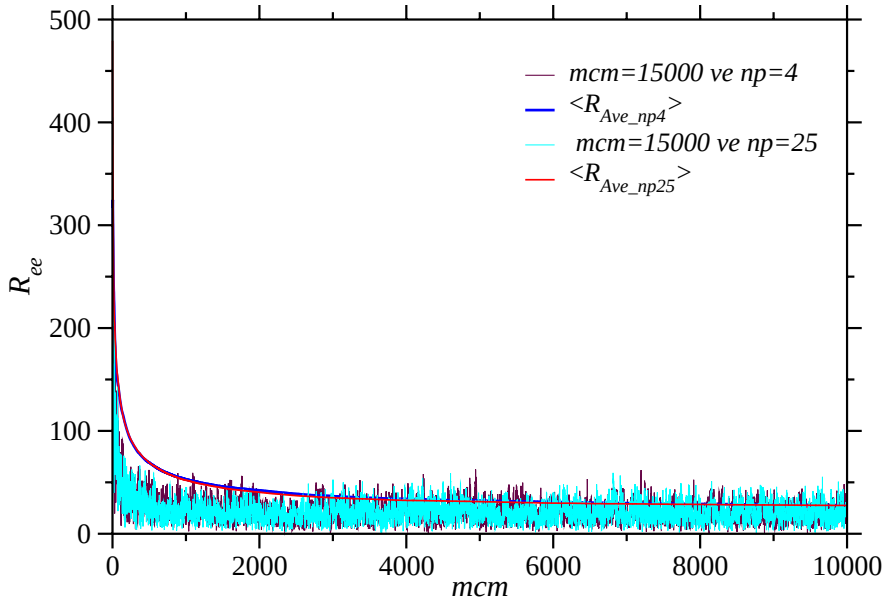
Şekil 4.8: Paralel hesapta N=480 iken mcm=5000 simulasyonunda sistemin bir denge durumuna eriştiğinin gözlenmesi.

ların yine "teorik öngörü" ile mükemmel uyumu görülmektedir.

4.5.3 Hızlanmalar ve kıyaslamalar

Yapılan paralel hesaplamalara ait göreli hızlanmalar Denklem 2.2'ye göre hesaplanılır. Tablo 4.1'de seri hesaba ait geçen süreler ve Tablo 4.2'deki paralel hesaba ait geçen süreler kullanılarak elde edilen göreli hızlanmalar Tablo 4.3'de listelenmektedir. Tablo 4.3'de görüldüğü üzere zincir uzunluğunun 480 olduğu durumda önemli bir göreli hızlanma kaydedilmemektedir. Yine de elde edilen göreli hızlanma işlemci sayısı ile artmakta fakat belli bir işlemci sayısından sonra hızlanma olmamaktadır. Hatta böyle bir durumda seri hesap tercihli duruma geçmektedir. Diğer yandan sistem büyüklüğü iki katına çıkartıldığında yani 960 olduğunda göreli hızlanmalar göze çarpmaya başlamaktadır. Bu göreli hızlanmaların ayrıca toplam simulasyon zamanına da bağlı olduğu gözlenmektedir.

Göreli hızlanmanın lineer olmayan davranışı teori ile uyum içerisindedir (Amdahl konusu için Bölüm 2.8.1'e bakınız). Zincir uzunluğu 480 ve 960 olan durumlara ait farklı simulasyon uzunluklarındaki göreli hızlanmaların grafikleri Şekil 4.16 ve Şekil 4.17'de verilmektedir. Bu grafiklerde gösterilen katı eğriler vektör üzerinde *lineer olmayan eğri*



Şekil 4.9: Paralel hesapta N=480 iken mcm=15000 simulasyonunda sistemin bir denge durumuna eriştiğinin gözlenmesi.

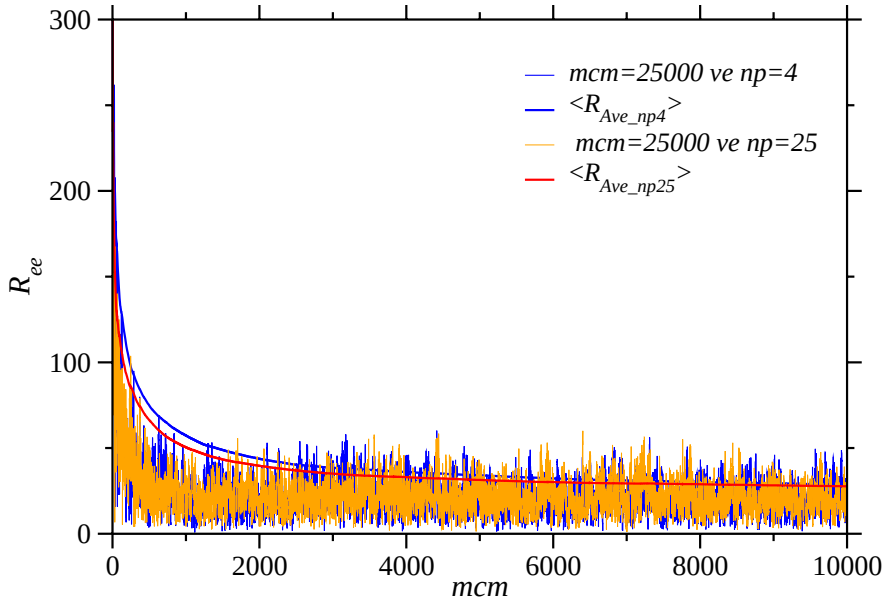
uydurma ("non-linear curve fitting") yöntemi ile elde edilmiştir[3].

Yapılan paralel hesap, seri işlenen kısmı (f) ve paralel işlenen kısmı ($1-f$) içermektedir. Tablo 4.4'de N=480 ve N=960 için zincir uzunluklarına ait paralel işlenen kısımların yüzdelikleri verilmektedir.

Diğer yandan paralel hesabın verimi ise Denklem 2.5 ile hesaplanır. Tablo 4.5'de verim değerleri verilmektedir.

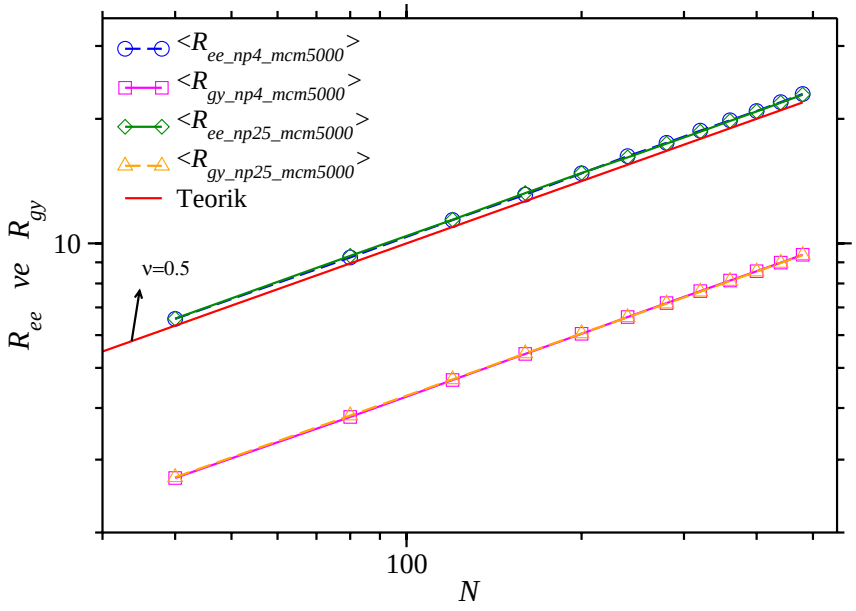
Tablodan gözüktüğü üzere hızlanmanın en yüksek olduğu durumda kaydedilen verim yaklaşık %14 civarındadır.

[3]Lineer olmayan eğri uydurmada kullanılan formül $y = A_0 x^{A_1} + A_2 x^{A_3}$ şeklindedir, burada $A_0 = -0,5769$, $A_1 = 0,4174$, $A_2 = 1,516$, $A_3 = 0,2359$ değerlerini almış olan başlangıç değişkenleridir.

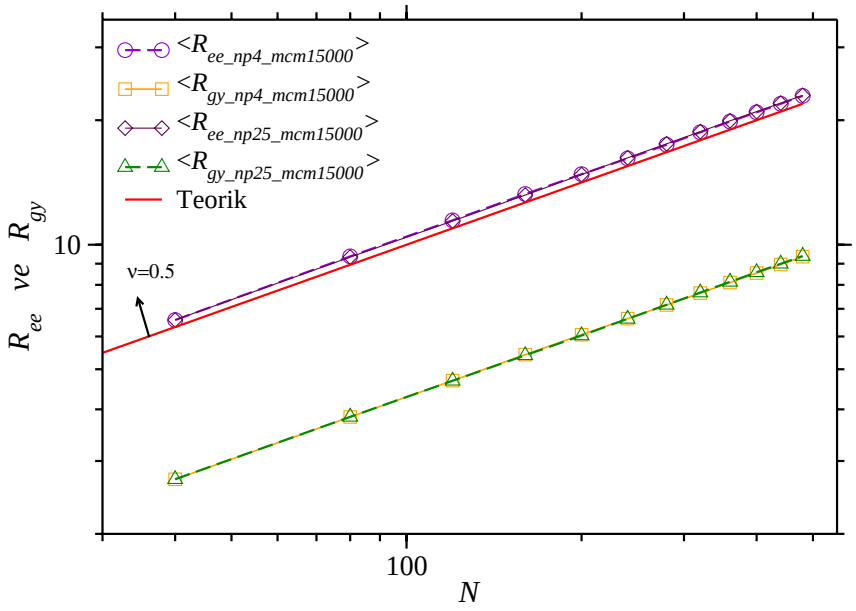


Şekil 4.10: Paralel hesapta N=480 iken mcm=25000 simulasyonunda sistemin bir denge durumuna eriştiğinin gözlenmesi.

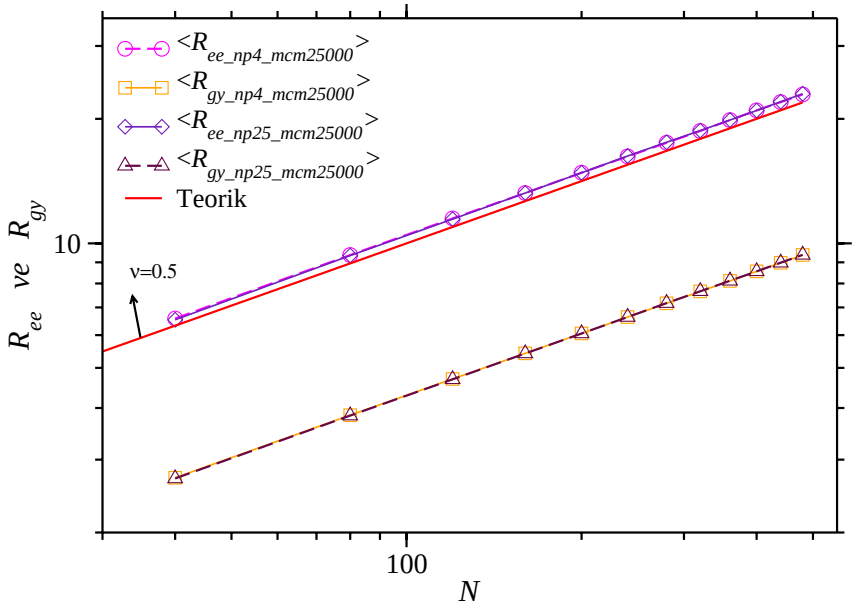
Tablo 4.4'den anlaşıldığı üzere ele alınan problemin paralelleştirilen kısmı %10 civarındadır. Programcı bakış açısından bu paralelleştirilen kısmın oranı arttırılmak istenir. Bunu yapabilmek için problemin içeriği detaylı olarak çalışılmalıdır. Problemdeki kısıtlamalar bu oranın azami mertebelerini belirlemektedir. Örneğin yukarıdaki yapılan paralel hesap yönetici makinenin en başta yaptığı dengeye gelme için gerekli hesaptan ve daha sonraki Monte Carlo adım hesaplarından sonraki kısma ait hızlanmalar üzerine odaklanalım (bu kısım için gerekli seri hesap süresi yaklaşık 815 s / 5000 *mcm* olarak ölçülmüştür). N=960 olan sistem için böyle bir değerlendirmede göreli hızlanma Şekil 4.18'de verilmektedir. Bu durumda göreli hızlanmanın en yüksek değeri 8 işlemcili durumda yaklaşık olarak 3 olarak bulunmuştur. Hesabın paralelleştirilebilen kısmı $(1 - f)$ %68 civarında olup elde edilen verim %39 civarındadır.



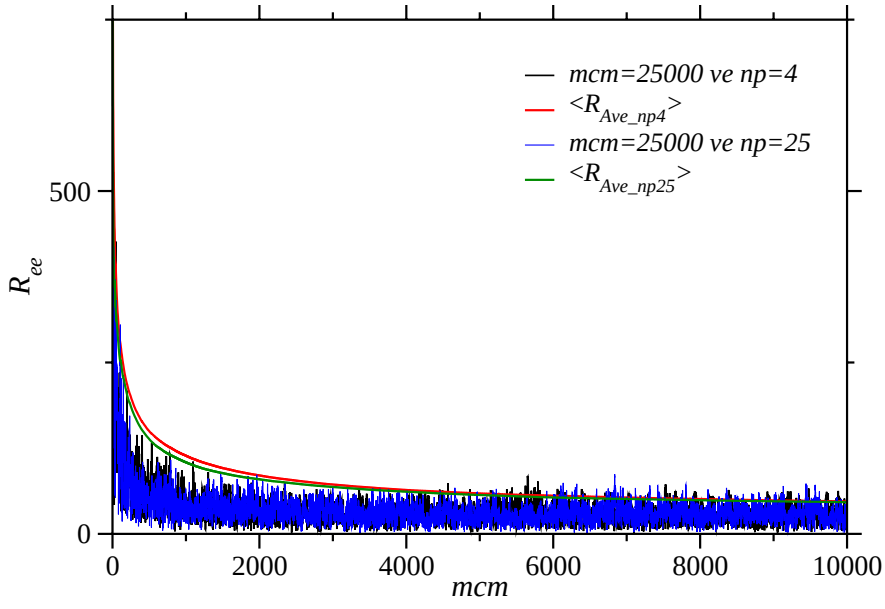
Şekil 4.11: Paralel hesapta N=480 iken mcm=5000 değerinde $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.



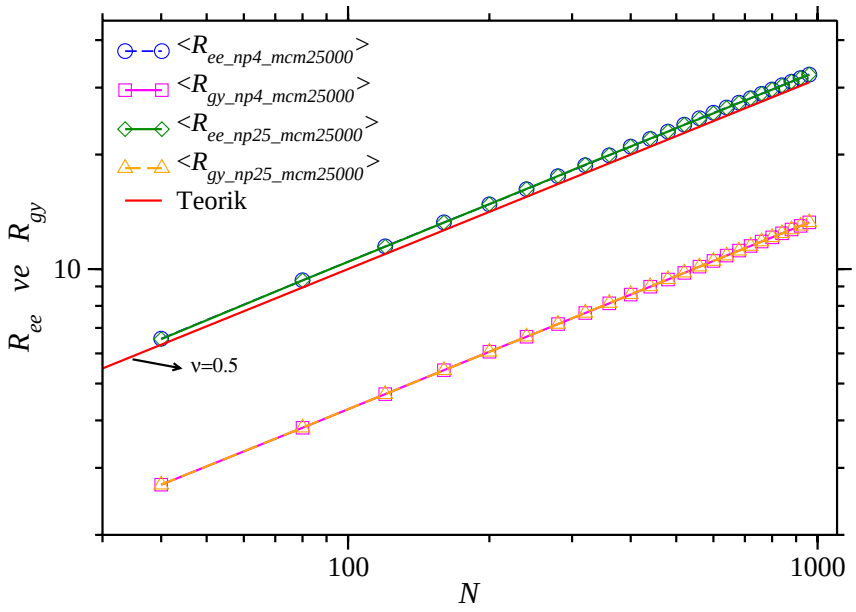
Şekil 4.12: Paralel hesapta N=480 iken mcm=15000 değerinde $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.



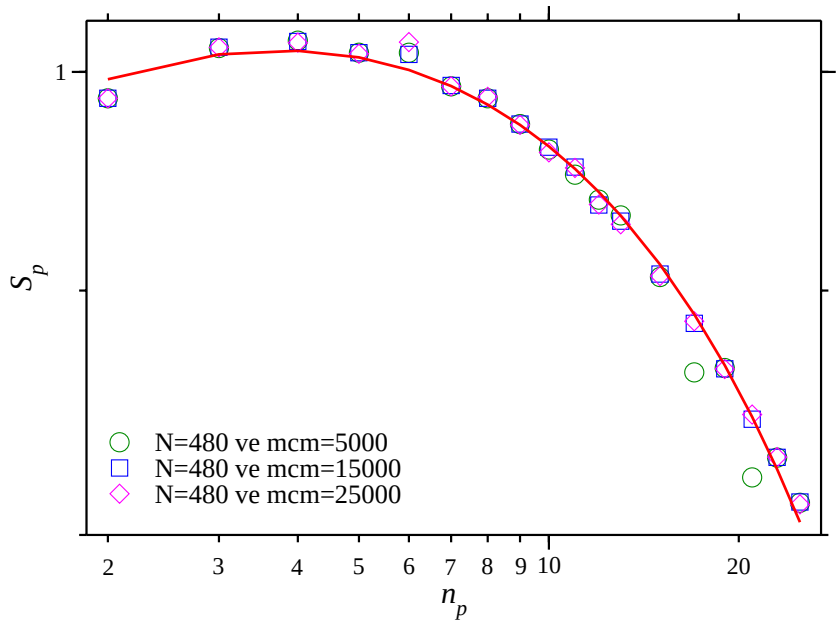
Şekil 4.13: Paralel hesapta N=480 iken mcm=25000 değerinde $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.



Şekil 4.14: Paralel hesapta N=960 iken mcm=25000 simulasyonunda sistemin bir denge durumuna eriştiğinin gözlenmesi..



Şekil 4.15: Paralel hesapta N=960 iken mcm=25000 değerinde $\langle R_{ee} \rangle$ ve $\langle R_{gy} \rangle$'nin N'e bağımlılığı.



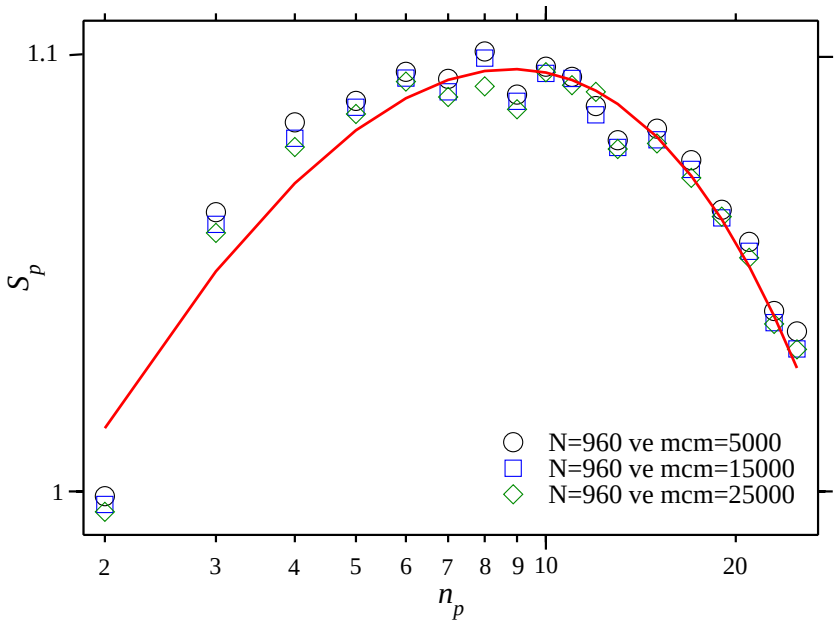
Şekil 4.16: Paralel hesapta N=480 iken üç farklı mcm değerinde hızlanma (S_p) ve işlemci sayısı (n_p)'ye ait grafik.

	N=480 için geçen süre (saniye)			**N=960 için geçen süre (saniye)**		
n_p	**mcm=5000**	**mcm=15000**	**mcm=25000**	**mcm=5000**	**mcm=15000**	**mcm=25000**
2	878	2635	4391	3744	11235	18730
3	857	2571	4285	3506	10531	17560
4	854	2564	4275	3434	10323	17214
5	859	2578	4298	3417	10249	17083
6	859	2580	4274	3394	10180	16955
7	873	2619	4366	3400	10213	17016
8	878	2635	4388	3378	10133	16975
9	889	2668	4448	3412	10235	17064
10	900	2698	4507	3390	10169	16919
11	911	2724	4541	3398	10181	16969
12	922	2774	4622	3421	10267	17807
13	929	2796	4666	3448	10345	17222
15	957	2868	4784	3439	10327	17200
17	1002	2937	4889	3464	10398	17338
19	1000	3002	5003	3504	10515	17494
21	1054	3076	5114	3530	10596	17661
23	1044	3133	5220	3587	10773	17933
25	1067	3201	5340	3604	10838	18039

Tablo 4.2: Paralel hesaba ait geçen süreler.

	N=480 için hızlanma oranı (S_p)			**N=960 için hızlanma oranı (S_p)**		
n_p	**mcm=5000**	**mcm=15000**	**mcm=25000**	**mcm=5000**	**mcm=15000**	**mcm=25000**
2	0.9874	0.9874	0.9874	0.9989	0.9970	0.9953
3	1.0116	1.0120	1.0119	1.0667	1.0637	1.0616
4	1.0152	1.0148	1.0142	1.0891	1.0851	1.0829
5	1.0093	1.0093	1.0088	1.0945	1.0929	1.0912
6	1.0093	1.0085	1.0145	1.1019	1.1003	1.0994
7	0.9931	0.9935	0.9931	1.1001	1.0968	1.0955
8	0.9874	0.9874	0.9881	1.1071	1.1054	1.0982
9	0.9752	0.9752	0.9748	1.0961	1.0944	1.0924
10	0.9633	0.9644	0.9620	1.1032	1.1015	1.1018
11	0.9517	0.9552	0.9548	1.1006	1.1002	1.0985
12	0.9403	0.9379	0.9381	1.0932	1.0910	1.0968
13	0.9332	0.9306	0.9292	1.0846	1.0828	1.0824
15	0.9059	0.9072	0.9063	1.0875	1.0847	1.0838
17	0.8652	0.8859	0.8868	1.0796	1.0773	1.0752
19	0.8670	0.8667	0.8666	1.0673	1.0653	1.0656
21	0.8225	0.8459	0.8478	1.0594	1.0571	1.0555
23	0.8304	0.8305	0.8306	1.0426	1.0398	1.0395
25	0.8125	0.8128	0.8119	1.0377	1.0335	1.0334

Tablo 4.3: Göreli hızlanma oranları.



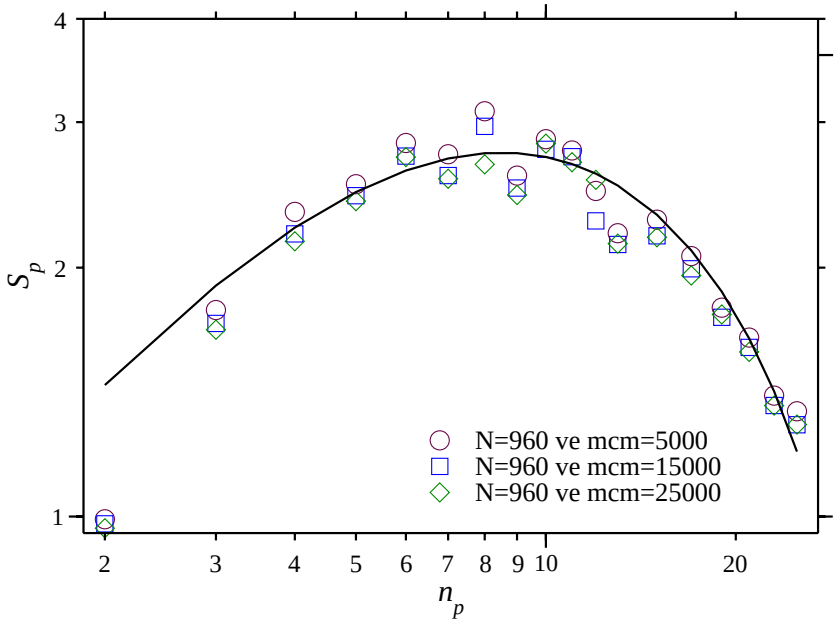
Şekil 4.17: Paralel hesapta N=960 iken üç farklı mcm değerinde hızlanma (S_p) ve işlemci sayısı (n_p)'ye ait grafik.

	N=480 için $(1-f)$ **(%)**			**N=960 için** $(1-f)$ **(%)**		
n_p	**mcm=5000**	**mcm=15000**	**mcm=25000**	**mcm=5000**	**mcm=15000**	**mcm=25000**
2	0.01	0.01	0.01	0.01	0.01	0.01
3	1.15	1.20	1.20	6.26	5.99	5.81
4	1.50	1.50	1.50	8.19	7.85	7.66
5	1.00	1.00	1.00	8.64	8.51	8.36
6	1.00	1.00	1.40	9.25	9.12	9.05
7	0.01	0.01	0.01	9.10	8.83	8.72
8	0.01	0.01	0.01	9.68	9.54	8.92
9	0.02	0.02	0.02	8.77	8.63	8.46
10	0.02	0.02	0.02	9.36	9.22	9.24
11	0.03	0.03	0.03	9.15	9.11	8.97
12	0.03	0.03	0.03	8.53	8.35	8.83
13	0.04	0.04	0.04	7.81	7.65	7.62
15	0.14	0.14	0.14	8.05	7.81	7.74
17	0.15	0.15	0.15	7.38	7.18	7.00
19	0.15	0.15	0.15	6.31	6.13	6.16
21	0.21	0.21	0.21	5.61	5.41	5.26
23	0.20	0.20	0.20	4.09	3.83	3.80
25	0.19	0.19	0.19	3.64	3.25	3.24

Tablo 4.4: Paralel hesap için $(1-f)$ yüzdelikleri.

	N=480 için ε verim (%)			**N=960 için ε verim (%)**		
n_p	**mcm=5000**	**mcm=15000**	**mcm=25000**	**mcm=5000**	**mcm=15000**	**mcm=25000**
2	49.3	49.3	49.3	49.9	49.8	49.7
3	33.7	33.7	33.7	35.5	35.4	35.3
4	25.3	25.3	25.3	27.2	27.1	27.1
5	20.1	20.1	20.1	21.8	21.8	21.8
6	16.8	16.8	16.8	18.3	18.3	18.3
7	14.18	14.19	14.18	15.71	15.71	15.71
8	12.34	12.33	12.34	13.83	13.83	13.83
9	10.83	10.80	10.81	12.17	12.17	12.17
10	9.63	9.64	9.63	11.03	11.03	11.03
11	8.65	8.68	8.67	10.0	10.0	10.0
12	7.83	7.81	7.82	9.11	9.11	9.11
13	7.17	7.15	7.16	8.34	8.32	8.32
15	6.03	6.04	6.03	7.25	7.23	7.22
17	5.08	5.21	5.20	6.35	6.34	6.33
19	4.56	4.56	4.56	5.61	5.61	5.61
21	3.91	4.02	4.01	5.04	5.04	5.04
23	3.61	3.61	3.61	4.53	4.52	4.51
25	3.25	3.25	3.25	4.15	4.13	4.13

Tablo 4.5: Paralel hesap için ε verim yüzdelikleri.



Şekil 4.18: Paralel hesapta N=960 iken üç farklı mcm değerinde hızlanma (S_p) ve işlemci sayısı (n_p)'ye ait grafik. Bu hesapta özel olarak işçi makinalar üzerine dağıtılan kısmın göreli hızlanması değerlendirilmiştir.

Bölüm 5

Sonuçlar

Bu yüksek lisans tezindeki çalışma, paralel hesaplama tekniğinin kullanımının günümüzün karmaşık problemlerinin çözümünde ne kadar önemli ve geçerli olduğunu bir örnek üzerinde sergilemiştir. Bilim adamları ve özellikle yazılım mühendisleri daima daha büyük, daha karmaşık sistemlerin bilgisayar ortamında çözülmesi yollarını araştırmaktalar ya da mevcut sistemi daha kısa zamanlarda çözmeyi hedeflemektedirler. Bu amaçla hesaplama birimlerininin mimarileri üzerinde gelişmeler kaydedilmekte ve ayrıca uygun yazılımlar geliştirilmektedir. Ancak teknolojinin gelmiş olduğu noktada daha hızlı makinalar üretebilmek çok kolay olmamaktadır. Ayrıca bütün yatırımlar oldukça maliyetlidir. Bunun yerine ekonomik çok sayıdaki işlem biriminin eşzamanlı olarak tek bir problemin çözümü için çalıştırılmaları daha uygun bir çözüm olmaktadır. Gerek çevre bilimlerine ait gerekse biyolojik sistemlere ait problemlerin bir tarafta büyük önemleri diğer tarafta çok fazla karmaşıklıkları paralel hesaplamanın sunduğu imkanlara bizleri yönlendirmektedir.

Bu çalışmada yapıldığı üzere, mesaj geçme arayüzü açık paralel hesaplamada en yaygın yöntemlerden biridir. Bu iş için ise MPI kütüphanesi oldukça kullanışlıdır. MPI kütüphanesinin C veya Fortran dilindeki kullanımı oldukça anlaşılır düzeydedir.

Bu çalışmadan anlaşıldığı üzere, seri hesabı yapılan bir problemin paralel hesabı genellikle seri hesaba nazaran problem hakkında daha çok bilgi taşır. Seri hesabı paralel hale getirmeden önce problemin seri hesabı çok iyi etüd edilmelidir. En iyi paralel hesabın problemin tabiatını çok iyi bildikten sonra ortaya çıkacağı beklenmektedir. Bu sebeble programcılar problemin detaylarını çok iyi bilmeye ihtiyaç duyacaklardır. Ayrıca herbir problemin kendine has hususiyetleri olacağı not edilmelidir.

Paralelleştirme yapılırken seri hesap mümkün olduğunca küçük kısımlara bölünmelidir. Daha sonra bu küçük kısımlar arasındaki iletişimler tayin edilmelidir. İşlemciler arasında iletişimlerde ciddi zaman kayıpları olacağından ve ayrıca işlemciler herbir göreve başlarken zaman kaybedeceklerinden bu durum gözönüne alınarak küçük parçaların

en iyi gruplandırmalarına gidilmelidir. Gruplandırılmış veya küçük hesap kısımları, eşzamanlı olarak çalıştırılmak üzere işlemcilere atanırken gerekli ise işlemcilerin güçlerine göre işler verilmelidir. Eğer hesap içerisinde birbirine karışacağından endişe edilen veriler var ise uygun yerlerde işlemciler senkronize edilmelidir.

Yukarıda söylendiği üzere paralel hesapta harcanan süre içinde iletişimde kaybedilen zaman, işlemcilerin işi başlatmaları için geçen zaman ve işlemcilerin boşta beklerken harcadıkları zaman çok önemli olmaktadır. Bu sebeble paralel hesap işlemci sayısı ile eniyileştirilebilir. Bu noktada Amdahl kanununun çok optimistik olduğu tekrar not edilmelidir. Diğer yandan mevcut iletişimde kullanılan bilgisayar ağının kaliteside önemlidir.

Dördüncü bölümün sonunda gösterildiği üzere amaç paralel olarak yürütülen kısmın seri olarak yürütülen kısma göre daha fazla olmasıdır. Paralel olarak yürütülen kısım ne kadar fazla ise göreli hızlanma da o kadar fazla olmaktadır.

Ek A

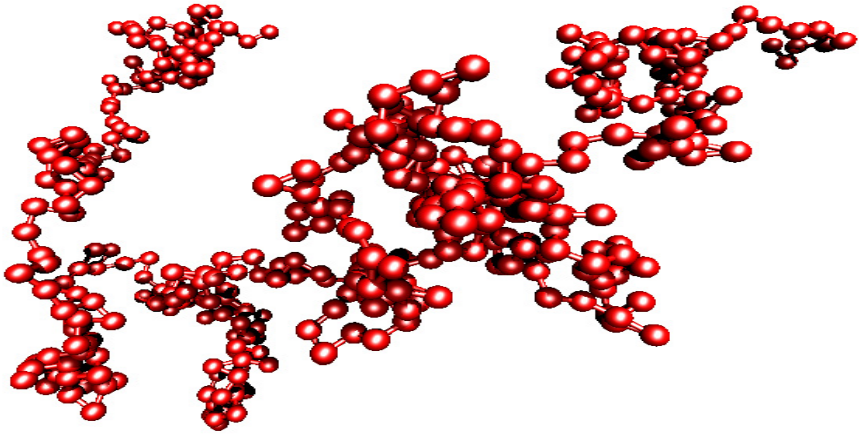
Sistemin anlık görüntüleri

Bu bölümde tez içerisinde tezin akışını bozmamak için detayları verilmemiş olan ideal polimer zinciri sisteminin N=480 ve N=960 durumlarında sistem dengede iken alınan anlık görüntüleri verilmektedir. Anlık görüntülerden de anlaşılacağı gibi polimer zincirleri teorik olarak öngörüldüğü biçimde kıvrılmış kangal yapıda (**"coil"**) gözlenmektedirler.

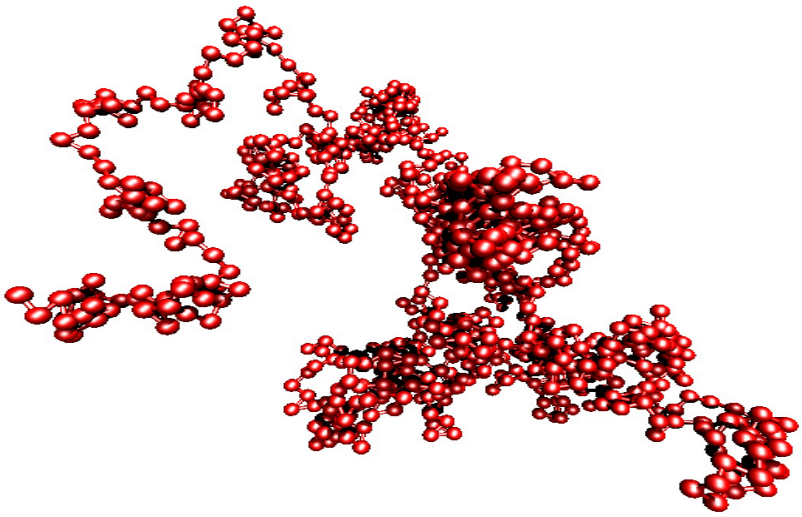
A.1 Anlık görüntüler

Aşağıda Şekil A.1'de N=480 için ve Şekil A.2'de N=960 için iki farklı mcm durumunda sistem dengede iken alınmış anlık görüntüler görülmektedir.

Bu anlık görüntüler simulasyon programının paralel olarak çalıştırılması sonucu üretilen konfigürasyon dosyası (monomerlerin yerleşiminin x, y ve z eksenine göre listelendiği veri dosyası) yardımıyla VMD (Visual Molecular Dynamics) programı yoluyla görselleştirilmiştir[53].



Şekil A.1: N=480 ve mcm=1000 durumunda iken sistemin anlık görüntüsü



Şekil A.2: N=960 ve mcm=3500 durumunda iken sistemin anlık görüntüsü

Ek B

MPI programlarının LAM/MPI ortamında derlenmesi ve çalıştırılması

Bu bölümde tez içerisinde tezin akışını bozmamak için detayları verilmemiş olan MPI programlarının LAM/MPI ortamında derlenmesi ve çalıştırılması konusu detaylı bir şekilde incelenecektir.

B.1 LAM/MPI'ın başlatılması ve bitirilmesi

LAM/MPI için oturum yönetimi konusuna bakacak olursak:

- Bir grup makineyi LAM'da çalıştırmak için,
  `lamboot -v <makine_listesi>`
- LAM'ı sonlandırmak için,
  `wipe -v <makine_listesi>`
  veya
  `lamhalt -v <makine_listesi>`
- LAM üzerindeki makinelerin düzgün çalışıp çalışmadığını görmek için,
  `recon -v <makine_listesi>`
- Süreçlerin anlık durumunu görmek için,
  `mpitask`
- Tüm süreçleri ve mesajları temizlemek için,
  `lamclean -v`

şeklinde komutlar ile çalışıldığını görürüz. Burada `makine_listesi` terimi önceden programcı tarafından hazırlanmış ve liste halinde LAM ortamında hesaplamaya katılması planlanarak boot ettirilmiş makinelerin makine isimlerini ve gerekiyorsa IP adreslerini içeren bir listedir. Bu liste sayesinde LAM/MPI grubuna katılan makineler ifade edilmiş olur.

B.2 LAM/MPI ile C ve Fortran programlarının derlenmesi ve çalıştırılması

Bir paralel C kodunu veya Fortran kodunu LAM/MPI ortamında derleyebilmek ve çalıştırabilmek için aşağıdaki komutlar kullanılır:

- Bir LAM/MPI programını derlemek için, C dilindeki bir program için:
  `mpicc -o <ikilidosya.out><kaynak.c> -I<include_dizini>`
  `-L<library_dizini> -l<kutuphane> -lmpi`
- Fortran dilindeki bir program için:
  `mpif77 -o <ikilidosya.out><kaynak.f> -I<include_dizini>`
  `-L<library_dizini> -l<kutuphane> -lmpi`
- Bir SPMD uygulaması başlatmak için,
  `mpirun -np<surec_sayisi> <program> --<argümanlar>`
- Bir MIMD uygulaması başlatmak için,
  `mpirun -v <uygulama_dosyasi>`

komutları komut satırından (uçbirim içerisinden) verilmelidir.

Ek C

Simulasyon programı kaynak kodu

Bu bölümde tez içerisinde tezin akışını bozmamak için detayları verilmemiş olan **ANSI (GNU) C** dilindeki simulasyon için geliştirdiğimiz seri ve paralel kod ayrı ayrı bölümler halinde verilmektedir.

C.1 Seri hesap yaptıran C kaynak kodu

```
//////////////////////////////////////////////////////////////////////
/////////     GNU C   ---  MAIN.C  coded by Bahadir Karasulu  //////////
////////////////////////////////// J U N E.2006 ////////////////////////
//////////////////////////////////////////////////////////////////////
///////////   M A I N  -  S I N G L E  C P U   v1.0 build 1000  ////////
//////////////////////////////////////////////////////////////////////


// Kullanilanacak Header dosyalari tanimlaniyor...
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <float.h>
#include <sys/select.h>
#include <sys/time.h>

////////////////// GENELE ACIK (PUBLIC) DEGISKENLER  ///////////////////

unsigned long mcm;
unsigned long nummon;
unsigned long chlencyc;
unsigned long relaxtime;

double angpick;
double picked;
unsigned long monopick;
double yer;
double transx,transy,transz,sx,sy,sz;
```

```
// Dosyalar tanimlaniyor...
FILE *dosya1,*dosya2,*dosya3,*dosya4,*dosya5,*dosya6,
     *dosya7,*dosya8,*dosya9,*dosya10;

int mcount,il;
double fact;
double xcomp,ycomp,zcomp;
double ree2,sree2[30],rgy2,srgy2[30];
int yenisay;
float SGyration;
int u,w;
unsigned long formul;

unsigned long mikrosaniye=0L, saniye=0L, dakika=0L, saat=0L;
double gecensure;
int namelen,cycle;
double x[1000],y[1000],z[1000],xx[1000],yy[1000],zz[1000];
char isim1[60];
char isim2[60];
char isim3[60];
char isim4[60];

int DosyaAc() {

///DOSYALARDA WRITE MODU...

/// REE_NOTEQUIL.TXT icin dosya tanimlanmasi ve dosyanin kayit blok konumlandirmalari...
if ((dosya3=fopen("Ree_NotEquil.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// iNFO.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya8=fopen("info.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_REE_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya9=fopen("Ree_vs_N.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_RGY_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya10=fopen("Rgy_vs_N.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}

/// DOSYALARDA APPEND MODU...

/// REE_NOTEQUiL.TXT icin dosya tanimlanmasi ve dosyanin kayit blok konumlandirmalari...
if ((dosya3=fopen("Ree_NotEquil.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// iNFO.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
```

```
if ((dosya8=fopen("info.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_REE_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya9=fopen("Ree_vs_N.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_RGY_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya10=fopen("Rgy_vs_N.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim Girdi Parametrelerini - Okuma kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int read_inputs(char dosya_ismi1[]) {

//Dosya actim, var mi? yok mu? kontrol ettim blogu --- TEXT DOSYA
if ((dosya1=fopen(dosya_ismi1,"r"))==NULL)   {
  printf("Dosya Bulunamadi...\n\n");
  exit(1);
}

fseek(dosya1,sizeof(int),1);
fscanf(dosya1,"%s\n%ld\n%s\n%ld\n%s\n%ld\n%s\n%ld\n",
        &isim1,&nummon,&isim2,&mcm,&isim3,&chlencyc,&isim4,&relaxtime);

//Kisaca Burada  NUMMON  ve  MCM degerleri  input_params.datas  dosyasindan
okunmus oldu...

fclose(dosya1);

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim  Koordinat - Okuma kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int read_coord(char dosya_ismi2[]) {
unsigned long i,say;
float xsub[nummon+1],ysub[nummon+1],zsub[nummon+1];
///// XSUB,YSUB,ZSUB => Koordinatlari yedekleyebilmek icin sabit uzunluklu diziler... /////


//Dosya actim, var mi? yok mu? kontrol ettim blogu --- TEXT DOSYA
if ((dosya2=fopen(dosya_ismi2,"r"))==NULL)   {
  printf("Dosya Bulunamadi...\n\n");
  exit(1);
```

```
}

fseek(dosya2,sizeof(float),1);

for (i=0;i<=nummon-1;i++)  {
   fscanf(dosya2,"%f       %f       %f\n",&xsub[i],&ysub[i],&zsub[i]);
   x[i]=xsub[i];
   y[i]=ysub[i];
   z[i]=zsub[i];
}

fclose(dosya2);

return 0;
}

////////////////////////////////////////////////////////////////////////////
//////
/* Bu Kisim PIVOT kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int pivot() {

int g=0;

//Rasgele Sayilar kumesi olusturuluyor...
picked = (double)rand()/RAND_MAX;
// Rasgele uretilen sayilar kumesinin icinden yalnizca bir tane sec ve elinde tut...//
// Anlik olarak rasgele secilmis daha onceden rasgele elde edilmis parcacik //
// genel formatta "alinmis parcacik" icin picked demistik,
// daha sonra bunun ismini "yer" olarak degistirdik.

yer=picked;
monopick=rint(yer*(nummon-1))+1;

/// rasgele secim kismi bitti ///


//////////////////////////  PIVOT MOVE  ////////////////////////////
// bu parcacigi orijin olarak dusunelim

transx=x[monopick];
transy=y[monopick];
transz=z[monopick];

// 1 inci parcadan Monopick e kadar...                  (1,monopick]
for (g=0;g<=monopick;g++) {  // yeni bir gecici xx, yy, zz arrayi olustur
   xx[g]=x[g];
   yy[g]=y[g];
   zz[g]=z[g];
}

// Monopick nolu parcacikdan sonuncu parcaciga kadar...   (monopick,nummon]

for (g=monopick+1;g<=nummon-1;g++) {
   xx[g]=x[g]-transx;
```

```
   yy[g]=y[g]-transy;
   zz[g]=z[g]-transz;
}


//////////////////    PIVOT MOVE --- DONDURME HAREKETLERI    ////////////////////////


////// Birinci KEZ DONDURME ///////
// Z ekseni boyunca donme hareketi yapsin...
picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sy=yy[g];
   xx[g]=sx*cos(angpick)+sy*sin(angpick);
   yy[g]=-(sx*sin(angpick))+sy*cos(angpick);
}

// Y ekseni boyunca donme hareketi yapsin...

picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sz=zz[g];
   xx[g]=sx*cos(angpick)+sz*sin(angpick);
   zz[g]=-(sx*sin(angpick))+sz*cos(angpick);
}


////// Ikinci KEZ DONDURME ///////
// Z ekseni boyunca donme hareketi yapsin...
picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sy=yy[g];
   xx[g]=sx*cos(angpick)+sy*sin(angpick);
   yy[g]=-(sx*sin(angpick))+sy*cos(angpick);
}

// Y ekseni boyunca donme hareketi yapsin...

picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
```

```
   sz=zz[g];
   xx[g]=sx*cos(angpick)+sz*sin(angpick);
   zz[g]=-(sx*sin(angpick))+sz*cos(angpick);
}

//////////////////////   DONDURME  HAREKETI SONU   ////////////////////////////////

for (g=monopick+1;g<=nummon-1;g++) {  //geri don
   xx[g]=xx[g]+transx;
   yy[g]=yy[g]+transy;
   zz[g]=zz[g]+transz;
}

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim METROPOLIS kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int metropolis(){

int g=0;

for (g=1;g<=nummon;g++) {
   x[g]=xx[g];
   y[g]=yy[g];
   z[g]=zz[g];

}
return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim END_TO_END kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int Rendtoend(int formul) {
xcomp=x[formul-1]-x[0];
ycomp=y[formul-1]-y[0];
zcomp=z[formul-1]-z[0];
ree2=xcomp*xcomp+ycomp*ycomp+zcomp*zcomp;
return 0;
}


////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim GYRATION kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int Gyration(int formul) {

u=0,w=0;
SGyration=0.0;
//  GYRATION subroutine kalibi...
for (u=0;u<=formul-1;u++) {
```

```
   for (w=0;w<=formul-1;w++) {
      xcomp=x[u]-x[w];
      ycomp=y[u]-y[w];
      zcomp=z[u]-z[w];
      SGyration=SGyration+((xcomp*xcomp)+(ycomp*ycomp)+(zcomp*zcomp));
   }
}
fact = 2.0*((double)formul*(double)formul);
rgy2=(double)(SGyration/fact);

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim ANA PROGRAM */
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char *argv[])
{
int kmax;
int i=0, j=0, g=0, k=0;
int say, sayac;
int m, n;
double SRunAve;
///  timevalue icin pointer atanmasi ve timezone atanmasi...
struct timeval t_ilk, t_son, t_dif;
struct timezone tz;
srand(time(0));
for (k=0;k<=30;k++){
   sree2[k]=0.0;
   srgy2[k]=0.0;
}
yenisay=0;
//Girdileri okumak icin ilgili dosyalari ac...
DosyaAc();
read_inputs("input_params.datas");
read_coord("config_to_start.datas");

//Relaxation Kismi...
SRunAve=0.0;
for (i=1;i<=relaxtime;i++){
   pivot();
   metropolis();
   Rendtoend(nummon);
   SRunAve+=ree2;
   fprintf(dosya3,"%d %f %f\n",i,sqrt(ree2),sqrt(SRunAve/(double)i));
}
for (k=1;k<=30;k++){
   sree2[k]=0.0;
   srgy2[k]=0.0;
}

//Saat tutmaya basla.
gettimeofday (&t_ilk, &tz);

for (i=1;i<=mcm;i++){
```

```
   for (j=0;j<=nummon-1;j++){
      mcount++;
      pivot();
      metropolis();
   }
   for (il=chlencyc;il<=nummon;il+=chlencyc){
      Rendtoend(il);
      Gyration(il);
      sree2[il/chlencyc]+=ree2;
      srgy2[il/chlencyc]+=rgy2;
   }
}

// Saat tutmayi bitir.
gettimeofday (&t_son, &tz);
saniye = t_dif.tv_sec = t_son.tv_sec - t_ilk.tv_sec;
gecensure=saniye;

kmax=nummon/chlencyc;
for (k=1;k<=kmax;k++){
   fprintf(dosya9,"%d %f\n",k*chlencyc,sqrt(sree2[k]/(double)mcm));
   fprintf(dosya10,"%d %f\n",k*chlencyc,sqrt(srgy2[k]/(double)mcm));
}

fprintf(dosya8,"...Program hakkinda bilgiler...\n%s\n%ld\n%s\n%ld\n%s\n%ld\n%s\n%ld\n",
        isim1,nummon,isim2,mcm,isim3,chlencyc,isim4,relaxtime);
fprintf(dosya8,"Gecen Sure saniye cinsinden=%f\n",gecensure);

//Dosyalari kapat...
fclose(dosya3);
fclose(dosya5);
fclose(dosya8);
fclose(dosya9);
fclose(dosya10);

exit(0);
}
```

C.2 Paralel hesap yaptıran C kaynak kodu

```
////////////////////////////////////////////////////////////////////////
/////////     GNU C   ---  MAIN.C  coded by Bahadir Karasulu  //////////
////////////////////////////////// J U N E.2006 ////////////////////////
////////////////////////////////////////////////////////////////////////
//////////   M A I N  -  M U L T I  C P U   v1.0 build 1000  //////////
////////////////////////////////////////////////////////////////////////


// Kullanilanacak Header dosyalari tanimlaniyor...
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```
#include <float.h>
#include <sys/select.h>
#include <sys/time.h>
#include "mpi.h"        ///  MPI (Message Passing Interface) Library Header'i...

/* Mesaj Etiketleri */
#define TAGA 100
#define TAGB 200
#define TAGC 300
#define TAGD 400
#define TAGE 500
#define TAGF 600

////////////////// GENELE ACIK (PUBLIC) DEGISKENLER   ///////////////////////

unsigned long mcm;
unsigned long nummon;
unsigned long chlencyc;
unsigned long relaxtime;

double angpick;
double picked;
unsigned long monopick;
double yer;
double transx,transy,transz,sx,sy,sz;

// Dosyalar tanimlaniyor...
FILE *dosya1,*dosya2,*dosya3,*dosya4,*dosya5,*dosya6,*dosya7,*dosya8,*dosya9,*dosya10;

int mcount,il;
double fact;
double xcomp,ycomp,zcomp;
double ree2,sree2[30],rgy2,srgy2[30];
int yenisay;
float SGyration;
int u,w;
unsigned long formul;

unsigned long mikrosaniye=0L, saniye=0L, dakika=0L, saat=0L;
double startwtime, endwtime,gecensure;
int namelen,cycle;
char processor_name[MPI_MAX_PROCESSOR_NAME];
double x[1000],y[1000],z[1000],xx[1000],yy[1000],zz[1000];
char isim1[60];
char isim2[60];
char isim3[60];
char isim4[60];

int DosyaAc() {

///DOSYALARDA WRITE MODU...

/// REE_NOTEQUIL.TXT icin dosya tanimlanmasi ve dosyanin kayit blok konumlandirmalari...
if ((dosya3=fopen("Ree_NotEquil.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
```

```
}
/// iNFO.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya8=fopen("info.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_REE_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya9=fopen("Ree_vs_N.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_RGY_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya10=fopen("Rgy_vs_N.dat","w"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}

/// DOSYALARDA APPEND MODU...

/// REE_NOTEQUiL.TXT icin dosya tanimlanmasi ve dosyanin kayit blok konumlandirmalari...
if ((dosya3=fopen("Ree_NotEquil.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// iNFO.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya8=fopen("info.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_REE_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya9=fopen("Ree_vs_N.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}
/// AVE_RGY_NUM.TXT icin ayrica bir dosyada tekli dokum ve dosyanin kayit blok konumlandirmalari...
if ((dosya10=fopen("Rgy_vs_N.dat","a+"))==NULL)   {
  printf("Hata : Dosya Olusturulamadi...\n\n");
  exit(1);
}

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim Girdi Parametrelerini - Okuma kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int read_inputs(char dosya_ismi1[]) {

//Dosya actim, var mi? yok mu? kontrol ettim blogu --- TEXT DOSYA
if ((dosya1=fopen(dosya_ismi1,"r"))==NULL)   {
  printf("Dosya Bulunamadi...\n\n");
  exit(1);
}

fseek(dosya1,sizeof(int),1);
```

```
fscanf(dosya1,"%s\n%ld\n%s\n%ld\n%s\n%ld\n%s\n%ld\n",
       &isim1,&nummon,&isim2,&mcm,&isim3,&chlencyc,&isim4,&relaxtime);


// Kisaca Burada  NUMMON  ve  MCM degerleri  input_params.datas  dosyasindan okunmus oldu...

fclose(dosya1);

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim  Koordinat - Okuma kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int read_coord(char dosya_ismi2[]) {
unsigned long i,say;
float xsub[nummon+1],ysub[nummon+1],zsub[nummon+1];
///// XSUB,YSUB,ZSUB => Koordinatlari yedekleyebilmek icin sabit uzunluklu diziler... /////


//Dosya actim, var mi? yok mu? kontrol ettim blogu --- TEXT DOSYA
if ((dosya2=fopen(dosya_ismi2,"r"))==NULL)   {
  printf("Dosya Bulunamadi...\n\n");
  exit(1);
}

fseek(dosya2,sizeof(float),1);

for (i=0;i<=nummon-1;i++)  {
   fscanf(dosya2,"%f       %f       %f\n",&xsub[i],&ysub[i],&zsub[i]);
   x[i]=xsub[i];
   y[i]=ysub[i];
   z[i]=zsub[i];
}

fclose(dosya2);

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim PIVOT kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int pivot() {

int g=0;

//Rasgele Sayilar kumesi olusturuluyor...
picked = (double)rand()/RAND_MAX;
// Rasgele uretilen sayilar kumesinin icinden yalnizca bir tane sec ve elinde tut...//
// Anlik olarak rasgele secilmis ve daha onceden rasgele elde edilmis parcacik //
// genel formatta "alinmis parcacik" icin picked demistik,
// daha sonra bunun ismini "yer" olarak degistirdik.
```

```
yer=picked;
monopick=rint(yer*(nummon-1))+1;

/// rasgele secim kismi bitti ///


//////////////////////////  PIVOT MOVE  ////////////////////////////
// bu parcacigi orijin olarak dusunelim

transx=x[monopick];
transy=y[monopick];
transz=z[monopick];

// 1 inci parcadan Monopick e kadar...                  (1,monopick]
for (g=0;g<=monopick;g++) {  // yeni bir gecici xx, yy, zz arrayi olustur
   xx[g]=x[g];
   yy[g]=y[g];
   zz[g]=z[g];
}

// Monopick nolu parcacikdan sonuncu parcaciga kadar...   (monopick,nummon]
for (g=monopick+1;g<=nummon-1;g++) {
   xx[g]=x[g]-transx;
   yy[g]=y[g]-transy;
   zz[g]=z[g]-transz;
}


///////////////    PIVOT MOVE --- DONDURME HAREKETLERI    /////////////////


////// Birinci KEZ DONDURME ///////
// Z ekseni boyunca donme hareketi yapsin...
picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sy=yy[g];
   xx[g]=sx*cos(angpick)+sy*sin(angpick);
   yy[g]=-(sx*sin(angpick))+sy*cos(angpick);
}

// Y ekseni boyunca donme hareketi yapsin...

picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sz=zz[g];
   xx[g]=sx*cos(angpick)+sz*sin(angpick);
   zz[g]=-(sx*sin(angpick))+sz*cos(angpick);
}
```

```
    ////// Ikinci KEZ DONDURME ///////
// Z ekseni boyunca donme hareketi yapsin...
picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sy=yy[g];
   xx[g]=sx*cos(angpick)+sy*sin(angpick);
   yy[g]=-(sx*sin(angpick))+sy*cos(angpick);
}

// Y ekseni boyunca donme hareketi yapsin...

picked = (double)rand()/RAND_MAX;
yer=picked;
angpick=yer*2.0*M_PI;                   //// Math.h icindeki pi degeri
                                             kullanimi...
for (g=monopick+1;g<=nummon-1;g++) {
   sx=xx[g];
   sz=zz[g];
   xx[g]=sx*cos(angpick)+sz*sin(angpick);
   zz[g]=-(sx*sin(angpick))+sz*cos(angpick);
}

///////////////// DONDURME  HAREKETI SONU   ////////////////////////////////////////

for (g=monopick+1;g<=nummon-1;g++) {  //geri don
   xx[g]=xx[g]+transx;
   yy[g]=yy[g]+transy;
   zz[g]=zz[g]+transz;
}

return 0;
}

//////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim METROPOLIS kismi - Subroutine'i*/
//////////////////////////////////////////////////////////////////////////////////

int metropolis(){

int g=0;

for (g=1;g<=nummon;g++) {
   x[g]=xx[g];
   y[g]=yy[g];
   z[g]=zz[g];

}
return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim END_TO_END kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int Rendtoend(int formul) {
xcomp=x[formul-1]-x[0];
ycomp=y[formul-1]-y[0];
zcomp=z[formul-1]-z[0];
ree2=xcomp*xcomp+ycomp*ycomp+zcomp*zcomp;
return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim GYRATION kismi - Subroutine'i*/
////////////////////////////////////////////////////////////////////////////////

int Gyration(int formul) {

u=0,w=0;
SGyration=0.0;
//  GYRATION subroutine kalibi...
for (u=0;u<=formul-1;u++) {
   for (w=0;w<=formul-1;w++) {
      xcomp=x[u]-x[w];
      ycomp=y[u]-y[w];
      zcomp=z[u]-z[w];
      SGyration=SGyration+((xcomp*xcomp)+(ycomp*ycomp)+(zcomp*zcomp));
   }
}
fact = 2.0*((double)formul*(double)formul);
rgy2=(double)(SGyration/fact);

return 0;
}

////////////////////////////////////////////////////////////////////////////////
/* Bu Kisim ANA PROGRAM */
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char *argv[])
{
MPI_Status status;
int num, rank, numprocs, next, from,kmax;
int i=0, j=0, g=0, k=0;
int say, sayac;
int m, n;
double SRunAve;
///  timevalue icin pointer atanmasi ve timezone atanmasi...
struct timeval t_ilk, t_son, t_dif;
struct timezone tz;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Get_processor_name(processor_name,&namelen);
printf("Prog. Toplam %d islemci ile calisiyor ve Aktif Islemci no: %d  ve
        Makine adi : %s\n",numprocs,rank,processor_name);
```

```
srand(time(0));
for (k=0;k<=30;k++){
   sree2[k]=0.0;
   srgy2[k]=0.0;
}
yenisay=0;
if (rank==0)  {
  DosyaAc();
  read_inputs("input_params.datas");
  read_coord("config_to_start.datas");
  fprintf(dosya8,"numprocs=%d\n",numprocs);
  fprintf(dosya8,"myrank=%d\n",rank);

  //Relaxation Kismi
  SRunAve=0.0;
  for (i=1;i<=relaxtime;i++){
     pivot();
     metropolis();
     Rendtoend(nummon);
     SRunAve+=ree2;
     fprintf(dosya3,"%d %f %f\n",i,sqrt(ree2),sqrt(SRunAve/(double)i));
  }
  // Saat tutmaya basla.
  startwtime = MPI_Wtime();

  for (i=1;i<=mcm;i++){
     if ( i >(mcm+1))
       {
        MPI_Finalize();
exit(0);
       }
     for (say=1;say<=numprocs-1;say++){
MPI_Ssend(&nummon,1,MPI_INT,say,1+say,MPI_COMM_WORLD);
MPI_Ssend(&mcm,1,MPI_INT,say,2+say,MPI_COMM_WORLD);
        MPI_Ssend(&chlencyc,1,MPI_INT,say,3+say,MPI_COMM_WORLD);
MPI_Ssend(&i,1,MPI_INT,say,TAGA+say,MPI_COMM_WORLD);

     }
     for (j=0;j<=nummon-1;j++){
mcount++;
pivot();
metropolis();
     }
     for (say=1;say<=numprocs-1;say++){
MPI_Ssend(x,nummon,MPI_DOUBLE,say,TAGB+say,MPI_COMM_WORLD);
MPI_Ssend(y,nummon,MPI_DOUBLE,say,TAGC+say,MPI_COMM_WORLD);
MPI_Ssend(z,nummon,MPI_DOUBLE,say,TAGD+say,MPI_COMM_WORLD);
     }
  }
  for (say=1;say<=numprocs-1;say++){
     kmax=nummon/chlencyc;
     for (k=say;k<=kmax;k+=(numprocs-1)){
MPI_Recv(&sree2[k],1,MPI_DOUBLE,say,TAGE+say,MPI_COMM_WORLD,&status);
MPI_Recv(&srgy2[k],1,MPI_DOUBLE,say,TAGF+say,MPI_COMM_WORLD,&status);
     }
  }
```

```
}
else
    {
     for ( ; ; ){
MPI_Recv(&nummon,1,MPI_INT,0,1+rank,MPI_COMM_WORLD,&status);
MPI_Recv(&mcm,1,MPI_INT,0,2+rank,MPI_COMM_WORLD,&status);
        MPI_Recv(&chlencyc,1,MPI_INT,0,3+rank,MPI_COMM_WORLD,&status);
MPI_Recv(&i,1,MPI_INT,0,TAGA+rank,MPI_COMM_WORLD,&status);

MPI_Recv(x,nummon,MPI_DOUBLE,0,TAGB+rank,MPI_COMM_WORLD,&status);
MPI_Recv(y,nummon,MPI_DOUBLE,0,TAGC+rank,MPI_COMM_WORLD,&status);
MPI_Recv(z,nummon,MPI_DOUBLE,0,TAGD+rank,MPI_COMM_WORLD,&status);
kmax=nummon/chlencyc;
for (k=rank;k<=kmax;k+=(numprocs-1)){
   il=k*chlencyc;
   Rendtoend(il);
   Gyration(il);
   sree2[k]+=ree2;
   srgy2[k]+=rgy2;
}
if(i==mcm) break;
     }
     for (k=rank;k<=kmax;k+=(numprocs-1)){
MPI_Send(&sree2[k],1,MPI_DOUBLE,0,TAGE+rank,MPI_COMM_WORLD);
MPI_Send(&srgy2[k],1,MPI_DOUBLE,0,TAGF+rank,MPI_COMM_WORLD);
     }
    }
//Saat tutmayi bitir
endwtime = MPI_Wtime();
gecensure=endwtime-startwtime;

if(rank==0){
  for (k=1;k<=kmax;k++){
     fprintf(dosya9,"%d %f\n",k*chlencyc,sqrt(sree2[k]/(double)mcm));
     fprintf(dosya10,"%d %f\n",k*chlencyc,sqrt(srgy2[k]/(double)mcm));
  }

fprintf(dosya8,"...PARALEL Program hakkinda bilgiler...\n%s\n%ld\n%s\n%ld\n%s\n%ld\n%s\n%ld\n",
        isim1,nummon,isim2,mcm,isim3,chlencyc,isim4,relaxtime);
fprintf(dosya8,"Gecen Sure saniye cinsinden=%f\n",gecensure);

//Dosyalari kapat...
fclose(dosya3);
fclose(dosya5);
fclose(dosya8);
fclose(dosya9);
fclose(dosya10);
}
MPI_Finalize();

exit(0);
}
```

Kaynakça

[1] Grama A., Gupta A., Karypis G., ve Kumar V. (2003), *"An Introduction to Parallel Computing, Second Edition"*. Addison Wesley Publishing, ISBN 0-201-64865-2.

[2] Lawrence Livermore Ulusal Laboratuar'ı (Amerika) web sayfası (2006), "Parallel Tutorials". *(Çevrimiçi : http://www.llnl.gov/computing/tutorials/parallel_comp/)* .

[3] Karniadakis G. E. ve Kirby II R. M. (2005), *"Parallel scientific computing in C++ and MPI"*. Cambridge University Press, 70-80.

[4] El-Rewini H. ve Abd-El-Barr M. (2005), *"Advanced Computer Architecture and Parallel Processing"*. Wiley-Interscience, John Wiley and Sons Inc. Publishing, ISBN 0-471-46740-5.

[5] Duato J., Yalamanchili S., ve Ni L. (2003), *"Interconnection Networks"*. Morgan Kaufmann Publishers, ISBN 1-55860-852-4.

[6] Vikipedi Özgür Ansiklopedi (2006). *(Çevrimiçi : http://en.wikipedia.org/)* .

[7] Foster I. (2006), "Designing and Building Parallel Programs". *(Çevrimiçi : http://www-unix.mcs.anl.gov/dbpp/text/node28.html)* .

[8] Tannenbaum A. (2001), *"Modern Operating Systems 2/E"*. Vrije University and Amsterdam and The Netherlands and Prentice Hall, ISBN 0-13-031358-0.

[9] Amdahl kanunu hakkında Temple Üniversitesi makalesi (2006). *(Çevrimiçi : http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html)* .

[10] En iyi 500 Süper Bilgisayarı gösteren liste (2005). *(Çevrimiçi : http://www.top500.org/)* .

[11] BEOWULF paralel hesaplama kümesi resmi sitesi (2006). *(Çevrimiçi : http://www.beowulf.org)* .

[12] Amalsi G. S. ve Gottlied A. (1994), *"Highly Parallel Computing", (2. baskı)*. Benjamin/Cummins.

[13] AMDAHL Hesap Makinesi Javascript (2006). *(Çevrimiçi : http://perfsuite.ncsa.uiuc.edu/AmdahlCalc/)* .

[14] Shi Y. (1996), *"Reevaluating Amdahl's Law and Gustafson's Law"*. Temple University and Computer-Information Sciences Department, Philadelphia, USA.

[15] Snir M., Otto S., Huss-Ledermann S., Walker D., ve Dongarra J. (1995), *"MPI:The Complete Reference"*. MIT Press.

[16] Gropp W., Lusk E., Doss N., ve Skjellum A. (1996), *"A high-performance, portable implementation of the MPI message passing interface standard"*. Parallel Computing - MIT Press, 22, 789-828.

[17] Message Passing Interface Forum (1997), "MPI-2: Extensions to the Message-Passing Interface" University of Tennessee and Knoxville and Tennessee. *(Çevrimiçi : http://www.mpi-forum.org)* .

[18] Gropp W., Lusk E., ve Skjellum A. (1994), *"Using MPI: portable parallel programming with the message-passing interface"*. MIT Press, ISBN 0-262-57104-8.

[19] Leighton F.T. (1992), *"Introduction to Parallel Algorithms and Architectures"*. Morgan-Kaufman Press.

[20] Hwang K. ve Xu Z. (1998), *"Scalable Parallel Computing"*. McGraw-Hill.

[21] Gropp W., Lusk E., ve Thakur R. (1999), *"Using MPI-2: Advanced features of the message-passing interface"*. MIT Press, ISBN 0-262-057133-1.

[22] LAM takımı (2004-2006), LAM/MPI Parallel Computing. *(Çevrimiçi: http://www.lam-mpi.org/)* .

[23] Argonne National Lab. resmi MPI websitesi (2006). *(Çevrimiçi: http://www-unix.mcs.anl.gov/mpi/)* .

[24] MPICH2 resmi websitesi (2006). *(Çevrimiçi : http://www-unix.mcs.anl.gov/mpi/mpich/)* .

[25] Cluster Info Centre (2006). *(Çevrimiçi : http://www.buyya.com/cluster)* .

[26] Pacheco P. (1996), *"Parallel Programming With MPI"*. Morgan Kaufmann Publishers.

[27] Pfister G. (1998), *"In Search of Clusters'*. Prentice Hall.

[28] Roosta S. (2000), *"Parallel Processing and Parallel Algorithms: Theory and Computation"*. Springer Verlag.

[29] Dix T. ve Buyya R. (2000), "CSC433: Parallel Systems" Monash Üniversitesi. *(Çevrimiçi : http://www.buyya.com/csc433/)* .

[30] NCSA (2001), 'Introduction to MPI created by the PACS Training Group NCSA Access". *Board of Trustees of the University of Illinois* .

[31] Buyya R. (ed.) (1999), *"High Performance Cluster Computing: Systems and Architectures"*. Prentice Hall.

[32] Andrews G. (1999), *"Foundations of Multithreaded, Parallel, and Distributed Programming"*. Addison-Wesley Publishing.

[33] Buyya R. (ed.) (1999), *"High Performance Cluster Computing: Programming and Applications"*. Prentice Hall.

[34] Apon A. (2006), "CSCE 4253: Concurrent Computing". *(Çevrimiçi : http://csce.uark.edu/ aapon/courses/concurrent/)* .

[35] Hwang K. (1998), "EE557: Computer Systems Architecture". *(Çevrimiçi : http://www-classes.usc.edu/engr/ee-s/557h/)* .

[36] Hwang K. (1998), "EE657: Parallel Processing". *(Çevrimiçi : http://wwwclasses.usc.edu/engr/ee-s/657h/)* .

[37] George B. ve Thomas Jr. (2003), *"Thomas's Calculus"*. Addison Wesley Publishing, ISBN 0-321-11636-4.

[38] Doi M. (1996), *"Introduction to polymer physics"*. Clarendon Press, Oxford.

[39] Colbourn E. A. (1994), *"Computer Simulations of Polymers"*. Longman Scientific and Technical, Harlow.

[40] Diaconis P. ve Saloff-Coste L. (1998), "What do we Know About the Metropolis Algorithms?". *Journal of Computer and System Sciences* **57**, 20.

[41] Hayes B. (1998), “How to Avoid yourself - American Scientist Article”. *Computing Science-American Scientist* **86**, 4, 314.

[42] Metropolis N., Ulam S., ve Amer J. (1949). *Statistical Assoc.* pages 44–335.

[43] Metropolis N., Rosenbluth A., Rosenbluth M., Teller A., ve Teller E. (1953), “Equations of state calculations by fast computing machines”. *J. Chem. Phys.* **21**, 1087.

[44] Uyaver Ş. (2004), *“Simulation of annealed polyelectrolytes in poor solvents” Doktora Tezi*. Max-Planck Institue, Berlin-Potsdam, Almanya.

[45] Breaud P. (1999), *“Markov Chains, Gibbs Fields, Monte Carlo Simulation and Queues”*. Texts in Applied Mathematics, Springer-Verlag.

[46] Binder K. ve Heermann D.W (1988), *“Monte Carlo simulation in statistical physics”*. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo.

[47] W. Hastings (1970), *“Monte Carlo Sampling Methods Using Markov Chains and their Applications”*. Biometrika, 57, 97-109.

[48] Binder K. (1986), *“Monte Carlo Methods in Statistical Physics 2nd ed., edited by K. Binder”*. volume 7 of Topics Curr. Phys. Springer-Verlag, Berlin and Heidelberg.

[49] Müller-Krumbhaar H. ve Binder K. (1973). *J. Stat. Phys.* **8**, 1.

[50] Liu J. (2001), *“Monte Carlo Strategies in Scientific Computing (Springer Series in Statistics)”*. Springer-Verlag.

[51] Binder K. (1995), *“Monte Carlo and Molecular Dynamics Simulations in Polymer Sciene”*. Oxford Univ. Press, New York.

[52] Baschnagel J. ve Arkadaşları (2000). *Adv. Polym. Sci.* **152**, 42.

[53] Görsel Moleküler Dinamikler aracı (2006). *(Çevrimiçi : http://www.ks.uiuc.edu/Research/vmd/)* .

Dizin

Özgeçmiş

Bahadır Karasulu, 1980 yılında İstanbul'da doğdu. İlk öğrenimini Sakarya'da tamamladıktan sonra, İstanbul Kartal Anadolu Lisesi'ni (Almanca) 1998 yılında tamamlayarak Kocaeli Üniversitesi Fen Edebiyat Fakültesi Fizik Bölümü'nü kazandı. Fizik Bölümünden 2003 yılında mezun oldu. Çeşitli firmalarda yazılımcı olarak çalıştıktan sonra 2004 yılında girdiği Maltepe Üniversitesi Fen Bilimleri Enstitüsü'ne bağlı olarak Bilgisayar Mühendisliği Yüksek Lisans eğitimine başladı. İyi derecede Almanca ve İngilizce bilmektedir. Halen eğitimine devam etmektedir.